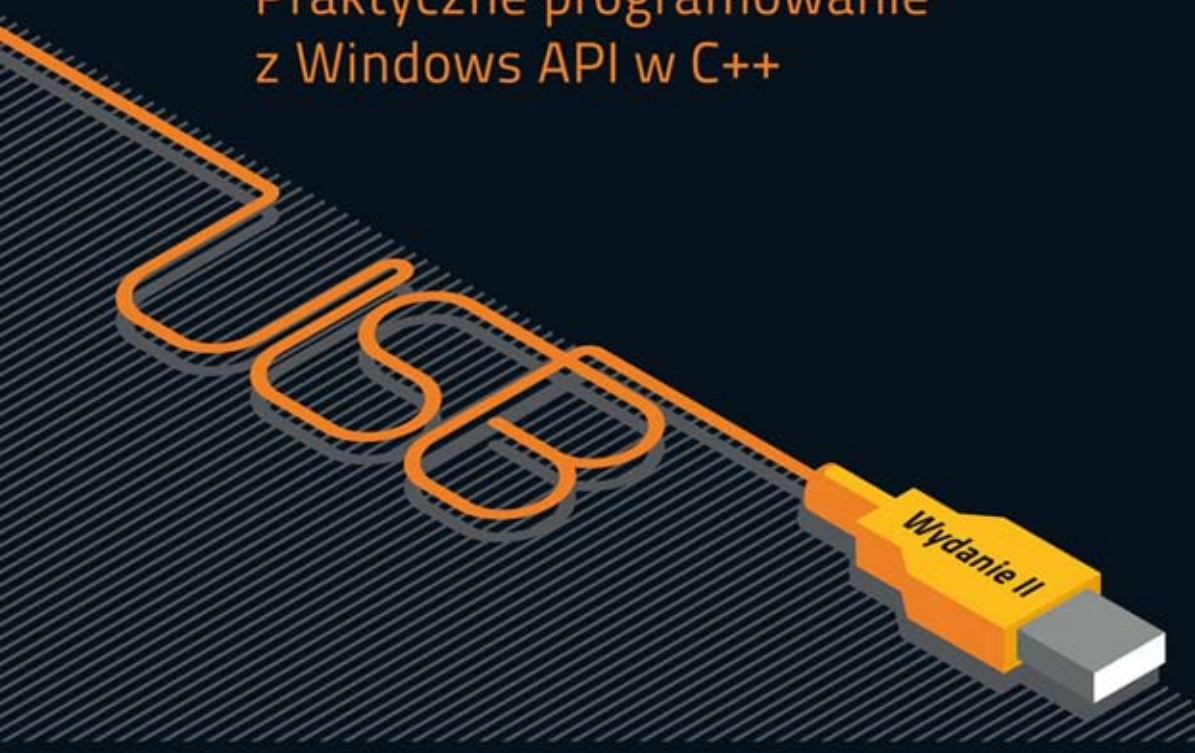
Andrzej Daniluk

USB

Praktyczne programowanie z Windows API w C++



USB dobre na wszystko — wykorzystaj jego moc!

- Standardy USB 2.0 i 3.0 oraz połączone urządzenia, czyli sprzętowa podstawa transmisji danych
- Transmisja danych w standardzie USB, czyli komunikacja i współdziałanie zasobów systemowych różnych urządzeń
- Biblioteki i programy wielowątkowe, czyli szczegółowe aspekty programowania transmisji danych w USB

Helion

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Redaktor prowadzący: Ewelina Burska
Projekt okładki: Studio Gravite/Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki
Materiały graficzne na okładce zostały wykorzystane za zgodą Shutterstock.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: http://helion.pl (księgarnia internetowa, katalog książek)

Kody źródłowe można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/usbpro.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie?usbpro
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

ISBN: 978-83-246-5539-7

Copyright © Helion 2013

Printed in Poland.

• Kup książkę
• Poleć książkę
• Oceń książkę

• Księgarnia internetowa
• Lubię to! » Nasza społeczność

Spis treści

Kup książkę Poleć książkę

Kup książkę Poleć książkę

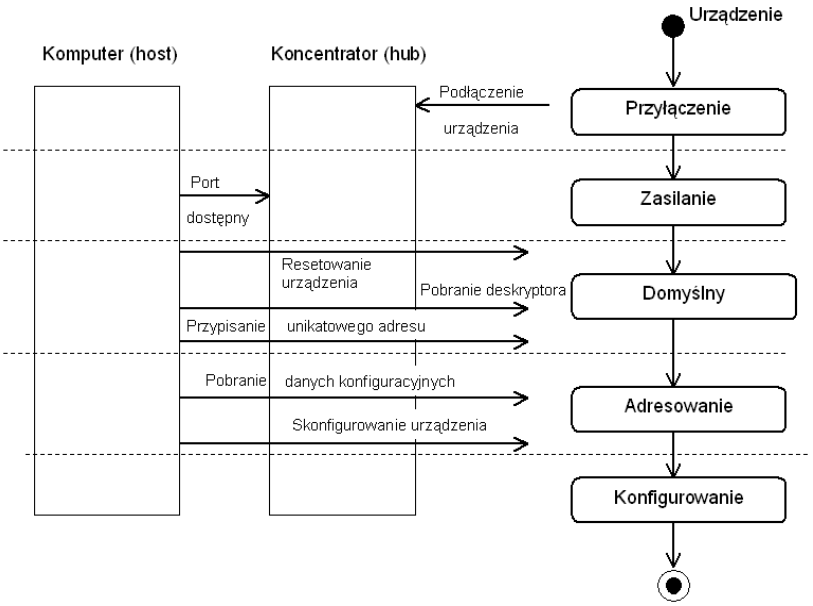
Kup książkę Poleć książkę

Kup książkę Poleć książkę

Rozdział 5.

Detekcja i identyfikacja urządzeń dołączonych do magistrali USB

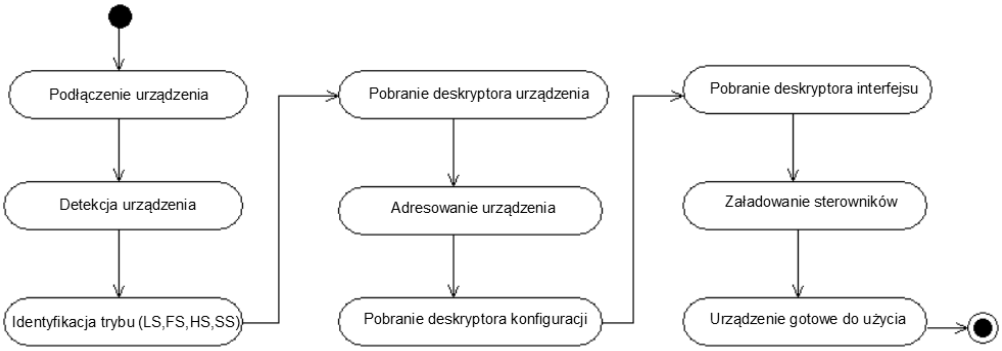
Urządzenia USB są automatycznie wykrywane przez system operacyjny po ich podłączeniu i włączeniu zasilania. Kiedy w systemie pojawi się nowy sprzęt, aktywowane są procedury jego detekcji i identyfikacji. Zespół tego typu operacji często jest określany jako wyliczanie lub enumeracja urządzeń (ang. *enumeration*). Rozpoczęcie procesu enumeracji powoduje przejście urządzenia między podstawowymi stanami, jak pokazano na rysunku 5.1.



Rysunek 5.1. *Podstawowe stany urządzenia w trakcie enumeracji*

Kup książkę Poleć książkę

Za pośrednictwem kilkunastu czynności, z których najważniejsze zostały przedstawione poniżej, system operacyjny wykonuje enumerację urządzenia w ramach poszczególnych stanów.



Rysunek 5.2. *Szczegółowy diagram czynności dla procesu enumeracji urządzeń dołączanych do magistrali USB*

- Użytkownik przyłącza urządzenie do portu USB hosta (macierzystego komputera) lub huba (koncentratora) — urządzenie pozostaje w stanie przyłączenia (ang. *attached state*).
- Po odblokowaniu linii zasilającej urządzenie przechodzi w stan zasilania (ang. *powered state*).
- Po sprawdzeniu stanu linii zasilających oprogramowanie hosta przystępuje do konfigurowania nowego sprzętu.
- Hub poprzez testowanie stanu linii sygnałowych sprawdza, z jaką prędkością przesyłu danych urządzenie może pracować. Informacja zostaje przekazana do hosta w odpowiedzi na wysłany przez niego rozkaz `GET_STATUS`.
- Kiedy nowe urządzenie zostaje rozpoznane, kontroler hosta wysyła do huba rozkaz `SET_FEATURE`. Port zostaje zresetowany (przez 10 ms linie sygnałowe pozostają w niskim stanie logicznym).
- Poprzez dalsze testowanie aktualnego stanu linii sygnałowych host sprawdza, czy urządzenie pracujące z pełną szybkością przesyłu danych może pracować też z szybkością wysoką.
- Ponownie wysyłając rozkaz `GET_STATUS`, host sprawdza, czy urządzenie pozostaje w stanie *Reset*. Jeżeli nie, zostaje utworzony potok zerowy przeznaczony do celów konfiguracji urządzenia. Urządzeniu zostaje przypisany domyślny adres `00h`, po czym przechodzi ono do stanu domyślnego (ang. *default state*).
- Host wysyła rozkaz `GET_DESCRIPTOR`, aby otrzymać informacje o maksymalnym rozmiarze pakietu danych, który może być transmitowany potokiem domyślnym. Rozkaz ten jest kierowany do zerowego punktu końcowego `EP0` urządzenia. Oprogramowanie hosta może identyfikować w danym czasie tylko jedno urządzenie, zatem tylko jedno urządzenie w danym czasie może pozostawać z adresem `00h`.

Kup książkę Poleć książkę

- W następnej kolejności za pośrednictwem rozkazu `SET_ADDRESS` urządzeniu jest przypisywany unikatowy adres — urządzenie przechodzi do stanu adresowania (ang. *addressed state*). Nowy adres pozostaje aktualny, dopóki urządzenie jest przyłączone do portu USB. W momencie odłączenia urządzenia port jest resetowany.
- W dalszej kolejności za pośrednictwem na nowo adresowanego rozkazu `GET_DESCRIPTOR` oprogramowanie hosta pobiera kompletne deskryptory urządzenia (patrz rysunek 3.9).
- Po odczytaniu deskryptorów urządzenia oprogramowanie hosta wyszukuje dla urządzenia najlepiej pasujący sterownik i zapisuje odpowiednie informacje (*Vendor ID*, *Product ID*, ...) w pliku *.inf*.
- Sterownik urządzenia wysyła rozkaz `SET_CONFIGURATION` w celu ostatecznego skonfigurowania nowego sprzętu. Od tego momentu urządzenie pozostaje w stanie skonfigurowania (ang. *configured state*)[1].

Podstawowe zasoby systemowe

Kompilator C++ w module *setupapi.h* udostępnia szereg użytecznych funkcji i struktur, które w znakomity sposób umożliwiają samodzielne przeprowadzenie detekcji i identyfikacji ścieżek dostępu do interfejsów oferowanych przez sterownik(i) urządzeń aktualnie dołączonych do magistrali USB. W tym podrozdziale zostały przedstawione najważniejsze z nich.

W dalszej części książki ze względu na zwięzłość sformułowań poprzez *interfejs urządzenia* będziemy rozumieli *interfejs, jaki sterownik urządzenia udostępnia warstwie aplikacji*.


Uwaga

Windows Driver Kit jest w pełni kompatybilny jedynie z kompilatorami VC++. W definicjach struktur i funkcji WDK w sposób niejednolity używa dla typów zmiennych rozszerzeń `IN` lub `__in` w celu oznaczenia parametrów wejściowych, `OUT` lub `__out` dla oznaczenia parametrów wyjściowych lub `__opt` dla oznaczenia parametrów opcjonalnych. Możliwe jest również występowanie oznaczeń będących kombinacją tych parametrów, np. `__inout` lub `__in__opt`. Niektóre kompilatory C++ mogą zgłaszać błędy w trakcie kompilacji modułów zawierających tego rodzaju oznaczenia w deklaracjach zmiennych. W przypadku napotkania przez kompilator problemów z używanymi przez WDK rozszerzeniami należy podjąć próbę zmiany ustawień opcji używanego kompilatora C++; można również bez jakiejkolwiek szkody dla oprogramowania opisane wyżej elementy, które nie są rozpoznawane przez kompilator, samodzielnie usunąć z odpowiednich plików nagłówkowych.

[1] Jeżeli w trakcie transmisji urządzenie USB 2.0 przez 3 ms nie wykrywa znacznika początku ramki danych *SOF*, przechodzi do stanu zawieszenia (ang. *suspended state*).

Kup książkę Poleć książkę

Funkcja SetupDiGetClassDevs()

Funkcja zwraca identyfikator klasy podłączonych urządzeń, których lista i opis konfiguracji znajduje się w rejestrze systemowym w kluczu HKEY_LOCAL_MACHINE (patrz rozdział 2.).

```
HDEVINFO
  SetupDiGetClassDevs(
    IN LPGUID  ClassGuid,  OPTIONAL
    IN PCTSTR  Enumerator,  OPTIONAL
    IN HWND  hwndParent,  OPTIONAL
    IN DWORD  Flags
    );
```

Parametr ClassGuid wskazuje strukturę GUID klasy urządzeń. Użycie tego parametru jest opcjonalne. Aplikacje użytkownika mogą pobierać adres identyfikatora GUID dla klasy urządzeń HID za pomocą funkcji HidD_GetHidGuid(). Wskaźnik Enumerator wskazuje łańcuch znaków (zakończony zerowym ogranicznikiem), przechowujący dane konkretnych urządzeń (patrz rozdział 2., rysunek 2.4). Użycie tego parametru w programie jest opcjonalne. Jeżeli wskaźnikowi przypiszemy wartość NULL, funkcja zwróci listę urządzeń typu PnP (ang. *Plug and Play*). Opcjonalnie wykorzystywany identyfikator hwndParent wskazuje okno odpowiedzialne za interakcję z otrzymanym zestawem urządzeń. Znacznik Flags przyjmuje postać bitowej alternatywy wybranego zestawu następujących stałych symbolicznych:

- DIGCF_ALLCLASSES — określa listę wszystkich zainstalowanych w systemie urządzeń;
- DIGCF_DEVICEINTERFACE — określa listę zainstalowanych urządzeń z danym interfejsem;
- DIGCF_DEFAULT — zwraca listę urządzeń z domyślnym interfejsem;
- DIGCF_PRESENT — określa urządzenia aktualnie dostępne w systemie;
- DIGCF_PROFILE — określa listę urządzeń będących częścią aktualnego zestawu sprzętowego.

Jeżeli wykonanie funkcji nie powiedzie się, zwracana jest wartość INVALID_HANDLE_VALUE. Kod ewentualnego błędu można zdiagnozować za pomocą funkcji GetLastError(). Szczegółowe kody błędów są dostępne na stronie *http://msdn.microsoft.com/en-us/library/windows/desktop/ms681382(v=vs.85).aspx*.

Funkcja SetupDiEnumDeviceInterfaces()

Funkcja wyszukuje interfejsy urządzeń identyfikowanych przez wskaźnik DeviceInfoSet zwracany przez funkcję SetupDiGetClassDevs().

```
WINSETUPAPI BOOL WINAPI
  SetupDiEnumDeviceInterfaces(
    IN HDEVINFO  DeviceInfoSet,
```

Kup książkę Poleć książkę

```
IN PSP_DEVINFO_DATA  DeviceInfoData,  OPTIONAL
IN LPGUID  InterfaceClassGuid,
IN DWORD  MemberIndex,
OUT PSP_DEVICE_INTERFACE_DATA  DeviceInterfaceData
);
```

Wskaźnik `DeviceInfoData` wskazuje strukturę `SP_DEVINFO_DATA` (patrz tabela 5.1), co umożliwia ograniczenie przeszukiwań istniejących urządzeń. Opcjonalnie funkcji można przekazać pusty wskaźnik. W takim wypadku funkcję należy wywoływać cyklicznie, tak aby przeszukała wszystkie interfejsy udostępniane przez sterownik danego urządzenia. Wskaźnik `InterfaceClassGuid` wskazuje strukturę `GUID`. Parametr wejściowy `MemberIndex` jest numerem odpytywanego interfejsu. Jego wartości zaczynają się od 0 (zerowy indeks pierwszego interfejsu — interfejsu domyślnego). Jeżeli funkcja jest wywoływana w pętli cyklicznie, przy każdym wywołaniu należy odpowiednio zwiększyć wartość `MemberIndex`. Jeżeli `SetupDiEnumDeviceInterfaces()` zwróci wartość `FALSE` oraz funkcja `GetLastError()` zwróci `ERROR_NO_MORE_ITEMS`, oznacza to, że nie znaleziono interfejsu o podanym indeksie. Wskaźnik `DeviceInterfaceData` wskazuje strukturę `SP_DEVICE_INTERFACE_DATA` (patrz tabela 5.2), której rozmiar należy wpisać do pola `cbSize`:

```
deviceInterfaceData.cbSize = sizeof(SP_DEVICE_INTERFACE_DATA);
```

Struktura SP_DEVINFO_DATA

W polach struktury są przechowywane informacje na temat egzemplarza urządzenia należącego do klasy urządzeń USB. W tabeli 5.1 zamieszczono jej opis.

Tabela 5.1. *Specyfikacja struktury SP_DEVINFO_DATA*

Typ	Element struktury	Znaczenie
`DWORD`	`cbSize`	Rozmiar struktury w bajtach
`GUID`	`ClassGuid`	Identyfikator `GUID` klasy urządzeń
`DWORD`	`DevInst`	Identyfikator wewnętrznej struktury opisującej urządzenie w systemie
`ULONG_PTR`	`Reserved`	Zarezerwowane

Windows Driver Kit (WDK) definiuje tę strukturę jako:

```
typedef struct _SP_DEVINFO_DATA {
  DWORD  cbSize;
  GUID  ClassGuid;
  DWORD  DevInst;
  ULONG_PTR  Reserved;
} SP_DEVINFO_DATA, *PSP_DEVINFO_DATA;
```

Definicja ta tworzy dwa nowe słowa kluczowe: `SP_DEVINFO_DATA` (struktura) i `PSP_DEVINFO_DATA` (wskaźnik do struktury).

Kup książkę Poleć książkę


Uwaga

Funkcje rodziny SetupDiXx(), używając struktury SP_DEVINFO_DATA jako parametru, automatycznie sprawdzają poprawność określenia jej rozmiaru. Aktualny rozmiar struktury należy wskazać za pomocą operatora sizeof() i wpisać do pola cbSize. Jeżeli rozmiar struktury w ogóle nie zostanie określony lub zostanie określony nieprawidłowo, to w przypadku użycia struktury jako parametru wejściowego IN zostanie wygenerowany błąd ERROR_INVALID_PARAMETER, natomiast przy korzystaniu ze struktury jako parametru wyjściowego OUT zostanie wygenerowany błąd ERROR_INVALID_USER_BUFFER.

Struktura SP_DEVICE_INTERFACE_DATA

Zasoby struktury SP_DEVICE_INTERFACE_DATA zaprezentowane w tabeli 5.2 przechowują dane interfejsu należącego do klasy urządzeń USB.

Tabela 5.2. *Specyfikacja struktury SP_DEVICE_INTERFACE_DATA*

Typ	Element struktury	Znaczenie
DWORD	cbSize	Rozmiar struktury w bajtach
GUID	InterfaceClassGuid	Identyfikator GUID interfejsu klasy urządzeń
DWORD	Flags	Znaczniki interfejsu. Wartość SPINT_ACTIVE oznacza, że interfejs jest aktualnie dostępny. Wartość SPINT_DEFAULT oznacza domyślny interfejs dla klasy urządzeń. Wartość SPINT_REMOVED określa usunięty interfejs
ULONG_PTR	Reserved	Parametr zarezerwowany i aktualnie nieużywany

WDK definiuje tę strukturę jako:

```
typedef struct _SP_DEVICE_INTERFACE_DATA {
  DWORD cbSize;
  GUID  InterfaceClassGuid;
  DWORD Flags;
  ULONG_PTR  Reserved;
} SP_DEVICE_INTERFACE_DATA, *PSP_DEVICE_INTERFACE_DATA;
```

Definicja ta tworzy dwa nowe słowa kluczowe: SP_DEVICE_INTERFACE_DATA (struktura) i PSP_DEVICE_INTERFACE_DATA (wskaźnik do struktury).


Uwaga

Funkcje zdefiniowane w module *setupapi.h*, używając struktury SP_DEVICE_INTERFACE_DATA jako parametru, automatycznie sprawdzają poprawność określenia jej rozmiaru. Aktualny rozmiar struktury należy wskazać za pomocą operatora sizeof() i wpisać do pola cbSize. Jeżeli rozmiar struktury w ogóle nie zostanie określony lub zostanie określony nieprawidłowo, system wygeneruje błąd ERROR_INVALID_USER_BUFFER.

Kup książkę Poleć książkę

Struktura SP_DEVICE_INTERFACE_DETAIL_DATA

Struktura SP_DEVICE_INTERFACE_DETAIL_DATA zawiera informacje o postaci ścieżki dostępu do interfejsu wybranego urządzenia USB. W tabeli 5.3 przedstawiono znaczenie poszczególnych pól tej struktury.

Tabela 5.3. *Specyfikacja struktury SP_DEVICE_INTERFACE_DETAIL_DATA*

Typ	Element struktury	Znaczenie
DWORD	cbSize	Rozmiar struktury w bajtach
TCHAR	DevicePath[ANYSIZE_ARRAY]	Łańcuch znaków zakończony zerowym ogranicznikiem (tzw. NULL — ang. *terminated string*), zawierający pełną nazwę symboliczną urządzenia (ścieżkę dostępu do interfejsu udostępnianego przez sterownik urządzenia). Parametr ten jest wykorzystywany przez funkcję CreateFile()

WDK definiuje tę strukturę jako:

```
typedef struct _SP_DEVICE_INTERFACE_DETAIL_DATA {
  DWORD  cbSize;
  TCHAR  DevicePath[ANYSIZE_ARRAY];
} SP_DEVICE_INTERFACE_DETAIL_DATA, *PSP_DEVICE_INTERFACE_DETAIL_DATA;
```

Definicja ta tworzy dwa nowe słowa kluczowe: SP_DEVICE_INTERFACE_DETAIL_DATA (struktura) i PSP_DEVICE_INTERFACE_DETAIL_DATA (wskaźnik do struktury).


Uwaga

Niekiedy ścieżkę dostępu do interfejsu urządzenia utożsamia się z jego nazwą symboliczną, którą można odczytać z rejestru systemowego (patrz rozdział 2.). Chociaż te dwa łańcuchy znaków mają bardzo podobną postać, to jednak mogą się różnić długością, dlatego w programach bezpieczniej jest posługiwać się kompletnymi danymi zapisanymi w polu DevicePath struktury SP_DEVICE_INTERFACE_DETAIL_DATA.

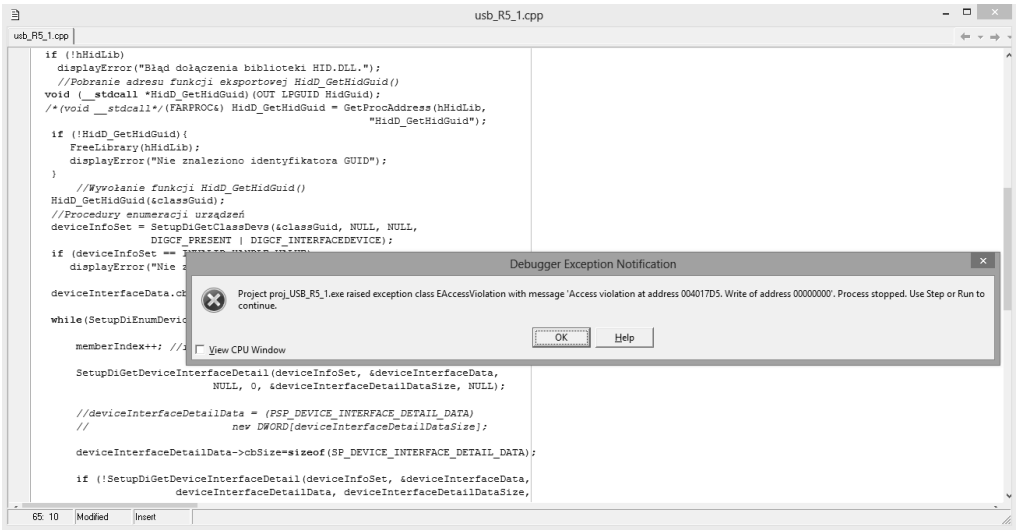
Funkcja SetupDiGetDeviceInterfaceDetail()

Funkcja zwraca szczegółowe informacje na temat interfejsu urządzenia.

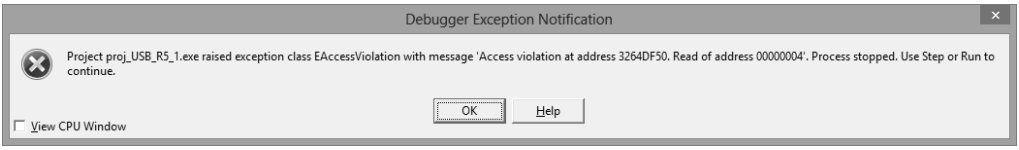
```
WINSETUPAPI BOOL WINAPI
  SetupDiGetDeviceInterfaceDetail(
    IN HDEVINFO  DeviceInfoSet,
    IN PSP_DEVICE_INTERFACE_DATA  DeviceInterfaceData,
    OUT PSP_DEVICE_INTERFACE_DETAIL_DATA  DeviceInterfaceDetailData, OPTIONAL
    IN DWORD  DeviceInterfaceDetailDataSize,
    OUT PDWORD  RequiredSize,  OPTIONAL
    OUT PSP_DEVINFO_DATA  DeviceInfoData  OPTIONAL
    );
```

Kup książkę Poleć książkę

Wskaźnik `DeviceInfoSet` jest zwracany przez funkcję `SetupDiGetClassDevs()`. Parametr `DeviceInterfaceData` wskazuje strukturę `SP_DEVICE_INTERFACE_DATA`. Wskaźnik `DeviceInterfaceDetailData` wskazuje strukturę `SP_DEVICE_INTERFACE_DETAIL_DATA` (patrz tabela 5.3); opcjonalnie zamiast niego do funkcji może być przekazana wartość `NULL`. W przypadku jawnego wskazania struktury `SP_DEVICE_INTERFACE_DETAIL_DATA` wskaźnik powinien być poprawnie zainicjowany, a jej pole `cbSize` musi być prawidłowo określone. W przeciwnym razie kompilator zgłosi błędy naruszenia pamięci, podobnie jak na rysunkach 5.3 i 5.4.



Rysunek 5.3. *Błąd naruszenia pamięci dla nieprawidłowo zainicjowanego wskaźnika do struktury SP_DEVICE_INTERFACE_DETAIL_DATA*



Rysunek 5.4. *Błąd naruszenia pamięci dla nieprawidłowo określonego rozmiaru pola cbSize struktury SP_DEVICE_INTERFACE_DETAIL_DATA*

Argument `DeviceInterfaceDetailDataSize` funkcji `SetupDiGetDeviceInterfaceDetail()` ma wartość zerową, jeżeli `DeviceInterfaceDetailData=NULL`; w przeciwnym razie określa rozmiar bufora: `(offsetof(SP_DEVICE_INTERFACE_DETAIL_DATA, DevicePath) + sizeof(TCHAR))`. Parametr `RequiredSize` jest wskaźnikiem do danej typu `DWORD`, której przypisuje się żądany rozmiar bufora wskazywanego przez `DeviceInterfaceDetailData`. Parametr `DeviceInfoData` jest wskaźnikiem do bufora danych przechowującego informacje na temat interfejsu udostępnianego przez sterownik urządzenia. Jeżeli wskaźnikowi nie przypisano wartości `NULL`, rozmiar danych powinien zostać określony za pomocą operatora `sizeof()`: `DeviceInfoData.cbSize=sizeof(SP_DEVINFO_DATA)`.

Kup książkę Poleć książkę

Funkcja SetupDiDestroyDeviceInfoList()

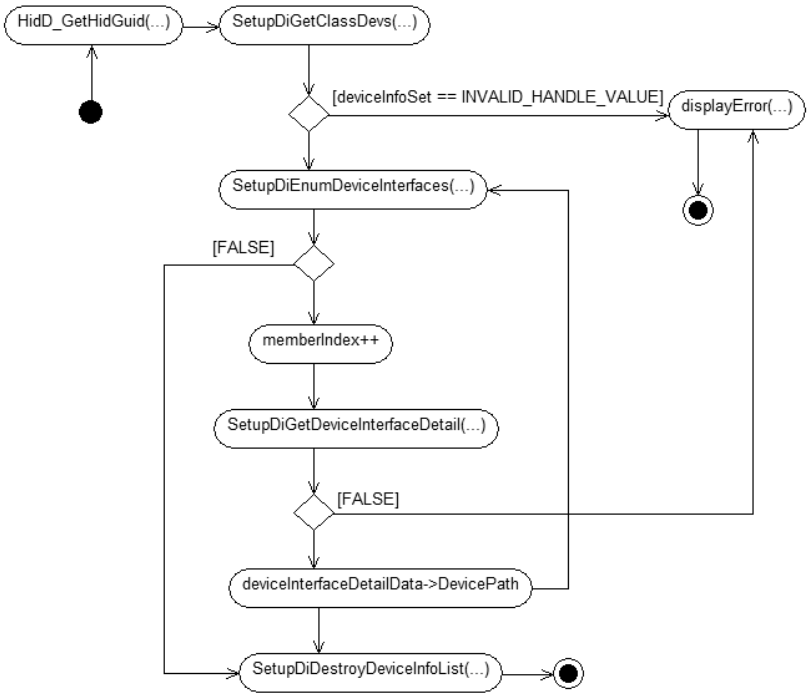
Funkcja usuwa wszystkie zaalokowane zasoby zawierające informacje o urządzeniu i zwalnia przydzieloną im pamięć. Kolejne urządzenia podłączane do systemu mogą korzystać ze zwolnionych zasobów.

```
WINSETUPAPI BOOL WINAPI
  SetupDiDestroyDeviceInfoList(
    IN HDEVINFO  DeviceInfoSet
    );
```

Wskaźnik `DeviceInfoSet` jest zwracany przez funkcję `SetupDiGetClassDevs()`. W przypadku prawidłowego zwolnienia zasobów funkcja zwraca wartość `TRUE`, w przeciwnym razie wartość `FALSE`. Kod wystąpienia błędu jest zwracany przez funkcję `GetLastError()`.

Detekcja interfejsów urządzeń

Na rysunku 5.5 w postaci diagramu czynności przedstawiono ogólną sieć działań, za pomocą których można programowo samodzielnie wykonać procedurę detekcji urządzeń klasy HID aktualnie podłączonych do systemu, co w efekcie powinno skutkować odzyskaniem *pełnych* nazw symbolicznych (pełnych ścieżek dostępu do interfejsów) urządzeń zapisanych w polu `DevicePath` struktury `SP_DEVICE_INTERFACE_DETAIL_DATA`.



Rysunek 5.5. *Ogólny diagram czynności dla operacji wstępnej enumeracji urządzeń klasy HID*

Czynności (w znaczeniu nadawanym przez UML) mogą być interpretowane w zależności od wybranej perspektywy: jako zestaw pojedynczych metod (z perspektywy projektowej) lub jako zadanie do wykonania, i to zarówno przez człowieka, jak i przez komputer (z perspektywy pojęciowej). Diagramów czynności można używać do opisu metod rozwiązywania problemów wyrażonych w postaci skończonej sekwencji kroków — to jest ich cel. Obsługują one wszystkie standardowe konstrukcje sterowania wymagane do opisania algorytmów [14].

W pierwszej kolejności należy odczytać postać identyfikatora `GUID` interfejsu klasy urządzeń występujących w systemie. Wskaźnik `deviceInfoSet` wskaże dane zawierające informacje na temat wszystkich zainstalowanych i aktualnie dostępnych (przyłączonych) urządzeń danej klasy. Następnie wyszukiwane są interfejsy poszczególnych urządzeń. Poprzez odczytanie zawartości pola `DevicePath` struktury `SP_DEVICE_INTERFACE_DETAIL_DATA` wydobywana jest pełna ścieżka dostępu `DevicePath` do interfejsu istniejącego urządzenia USB. Na koniec dotychczas używane przez program zasoby są zwalniane.

Niektóre z dostępnych kompilatorów języka C++ mogą niewłaściwie obliczać rozmiar struktur (za pomocą operatora `sizeof()`). Błędne obliczenie rozmiaru którejkolwiek z używanych struktur będzie niezmiennie skutkować błędami w trakcie uruchamiania programu. W takich sytuacjach należy zadbać o właściwe ustalenie opcji kompilatora na podstawie jego dokumentacji. Stosowana tu konstrukcja:

```
#pragma option push -a1
  //...
#pragma option pop
```

odpowiada opisanej sytuacji. Inne przykłady rozwiązania tego typu problemów można znaleźć w artykule dostępnym pod adresem: *http://support.codegear.com/article/35751*.

Na listingu 5.1 zamieszczono kod modułu projektu będącego uszczegółowioną implementacją diagramu z rysunku 5.5.

Listing 5.1. *Kod modułu usb_R5_1.cpp jako przykład zaprogramowania wstępnej enumeracji urządzeń na podstawie identyfikatora GUID klasy urządzeń*

```
#include <windows>
#pragma option push -a1
  #include <setupapi>
#pragma option pop
#include <assert>
#include <iostream>

using namespace std;

void displayError(const char* msg){
  cout << msg << endl;
  system("PAUSE");
  exit(0);
};
//---------------------------------------------------------
template <class T>
```

```
inline void releaseMemory(T &x)
{
  assert(x != NULL);
  delete [] x;
  x = NULL;
}
//-------------------------------------------------------
GUID classGuid;
HMODULE hHidLib;
DWORD /* unsigned long lub ULONG */ memberIndex = 0;
DWORD deviceInterfaceDetailDataSize;
DWORD requiredSize;

HDEVINFO deviceInfoSet;
SP_DEVICE_INTERFACE_DATA deviceInterfaceData;
PSP_DEVICE_INTERFACE_DETAIL_DATA deviceInterfaceDetailData = NULL;

int main(){
    //Odwzorowanie identyfikatora biblioteki HID.dll w przestrzeni
    //adresowej głównego procesu
  hHidLib = LoadLibrary("C:\\Windows\\system32\\HID.DLL");
  if (!hHidLib)
    displayError("Błąd dołączenia biblioteki HID.DLL.");
    //Pobranie adresu funkcji eksportowej HidD_GetHidGuid()
  void (__stdcall *HidD_GetHidGuid)(OUT LPGUID HidGuid);
  /*(void __stdcall*/(FARPROC&) HidD_GetHidGuid = GetProcAddress(hHidLib,
                                                    "HidD_GetHidGuid");
   if (!HidD_GetHidGuid){
      FreeLibrary(hHidLib);
      displayError("Nie znaleziono identyfikatora GUID.");
   }
   //Wywołanie funkcji HidD_GetHidGuid()
   HidD_GetHidGuid(&classGuid);
   //Procedury enumeracji urządzeń
   deviceInfoSet = SetupDiGetClassDevs(&classGuid, NULL, NULL,
                  DIGCF_PRESENT | DIGCF_INTERFACEDEVICE);
   if (deviceInfoSet == INVALID_HANDLE_VALUE)
      displayError("Nie zidentyfikowano podłączonych urządzeń.\n");

   deviceInterfaceData.cbSize = sizeof(SP_DEVICE_INTERFACE_DATA);

   while(SetupDiEnumDeviceInterfaces(deviceInfoSet, NULL, &classGuid,
                                  memberIndex, &deviceInterfaceData)){
       memberIndex++; //inkrementacja numeru interfejsu

       SetupDiGetDeviceInterfaceDetail(deviceInfoSet, &deviceInterfaceData,
                           NULL, 0, &deviceInterfaceDetailDataSize, NULL);

       deviceInterfaceDetailData = (PSP_DEVICE_INTERFACE_DETAIL_DATA)
                            new DWORD[deviceInterfaceDetailDataSize];

       deviceInterfaceDetailData->cbSize=sizeof(SP_DEVICE_INTERFACE_DETAIL_DATA);

       if (!SetupDiGetDeviceInterfaceDetail(deviceInfoSet, &deviceInterfaceData,
                    deviceInterfaceDetailData, deviceInterfaceDetailDataSize,
                    &requiredSize, NULL)){
           releaseMemory(deviceInterfaceDetailData);
```

```
            //SetupDiDestroyDeviceInfoList(deviceInfoSet);
            //displayError("Nie można pobrać informacji o interfejsie.\n");
        }
        // deviceInterfaceDetailData->DevicePath jest łączem symbolicznym
        // do interfejsu urządzenia
        cout << deviceInterfaceDetailData->DevicePath << endl;
        releaseMemory(deviceInterfaceDetailData);
    }; //koniec while
    SetupDiDestroyDeviceInfoList(deviceInfoSet);
    FreeLibrary(hHidLib);
    cout << endl;
    system("PAUSE");
    return 0;
}
//-------------------------------------------------------
```

Różne odmiany wskaźnika do funkcji FARPROC są zdefiniowane w module *windef.h*. W Linuksie FARPROC należy zastąpić wskaźnikiem ogólnym void*. Windows Driver Kit stosuje konwencję __stdcall, co zapewnia, że funkcja zostanie wywołana zgodnie z wymogami systemu operacyjnego. Oznacza to, że w funkcji wywołującej liczba i typ argumentów muszą być poprawne. Funkcje i typy danych definiowane w zasobach WDK API często korzystają z następujących makrodefinicji:

```
#define DDKAPI __stdcall
```

lub

```
#define DDKAPI_PTR __stdcall*
```

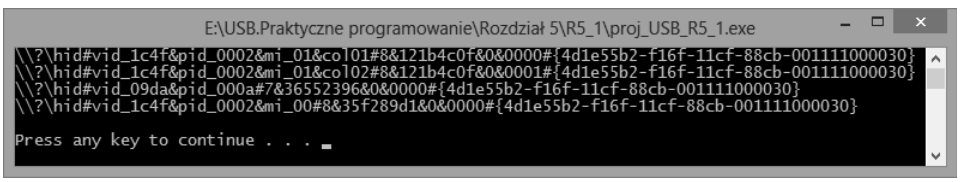
W tego typu konwencjach deklaracja wskaźnika do funkcji może zostać zapisana na jeden z dwóch sposobów:

```
void (DDKAPI *HidD_GetHidGuid)(OUT LPGUID HidGuid);
```

lub

```
void (DDKAPI_PTR HidD_GetHidGuid)(OUT LPGUID HidGuid);
```

Na rysunku 5.6 przedstawiono działający program z listingu 5.1. Wynik działania programu należy porównać z odpowiednimi zapisami w edytorze rejestrów (patrz rysunek 2.5).



Rysunek 5.6. *Aplikacja proj_USB_R5_1 w trakcie działania*

Odczytane ścieżki dostępu (łącza symboliczne) do poszczególnych interfejsów urządzeń klasy HID mogą być przekazane do funkcji CreateFile() w celu otrzymania identyfikatora pliku sterownika urządzenia wykonawczego USB, które jest aktualnie dostępne w systemie.

Zliczanie interfejsów urządzeń

Dokonując niewielkiej modyfikacji poprzednio prezentowanego algorytmu, można zbudować uniwersalną funkcję `searchInterfaceHidDevices()`, która dodatkowo zlicza interfejsy aktualnie podłączonych do systemu urządzeń klasy HID. Listing 5.2 zawiera odpowiedni przykład będący modyfikacją kodu z listingu 5.1.

Listing 5.2. *Kod modułu usb_R5_2.cpp*

```
#include <windows>
#pragma option push -a1
  #include <setupapi>
#pragma option pop
#include <assert>
#include <iostream>

using namespace std;

void displayError(const char* msg){
  cout << msg << endl;
  system("PAUSE");
  exit(0);
};
//---------------------------------------------------------
template <class T>
inline void releaseMemory(T &x)
{
  assert(x != NULL);
  delete [] x;
  x = NULL;
}
//---------------------------------------------------------
int searchInterfaceHidDevices()
{
    HMODULE hHidLib;
    HDEVINFO deviceInfoSet;
    SP_INTERFACE_DEVICE_DATA deviceInterfaceData;
    DWORD  memberIndex = 0;
    GUID classGuid;
    PSP_DEVICE_INTERFACE_DETAIL_DATA deviceInterfaceDetailData = NULL;
    DWORD requiredSize = 0;
    DWORD deviceInterfaceDetailDataSize = 0;
    DWORD searchMaxDevice = 100;
    bool done = false;

    void (__stdcall *HidD_GetHidGuid)(OUT LPGUID HidGuid);

    hHidLib = LoadLibrary("C:\\Windows\\system32\\HID.DLL");
    if (!hHidLib)
      displayError("Błąd dołączenia biblioteki HID.DLL.");

    (FARPROC&) HidD_GetHidGuid = GetProcAddress(hHidLib,
                                                "HidD_GetHidGuid");
    if (!HidD_GetHidGuid){
       FreeLibrary(hHidLib);
```

```
        displayError("Nie znaleziono identyfikatora GUID.");
    }

    HidD_GetHidGuid (&classGuid);

    deviceInfoSet = SetupDiGetClassDevs(&classGuid, NULL, NULL,
                                        (DIGCF_PRESENT | DIGCF_DEVICEINTERFACE));

    deviceInterfaceData.cbSize = sizeof(SP_INTERFACE_DEVICE_DATA);

    while(!done) {
       for(; memberIndex < searchMaxDevice; memberIndex++) {
          if(SetupDiEnumDeviceInterfaces(deviceInfoSet,0,&classGuid,
                                         memberIndex,&deviceInterfaceData)) {
            SetupDiGetDeviceInterfaceDetail(deviceInfoSet,&deviceInterfaceData,
                                         NULL,0,&deviceInterfaceDetailDataSize,
                                         NULL);
            requiredSize = deviceInterfaceDetailDataSize;
            deviceInterfaceDetailData = (PSP_DEVICE_INTERFACE_DETAIL_DATA)\
                              new DWORD[deviceInterfaceDetailDataSize];

               if(deviceInterfaceDetailData) {
                  deviceInterfaceDetailData->cbSize=\
                              sizeof(SP_INTERFACE_DEVICE_DETAIL_DATA);
                }
                else {
                   SetupDiDestroyDeviceInfoList(deviceInfoSet);
                   releaseMemory(deviceInterfaceDetailData);
                   return 0;
                }
                if(!SetupDiGetDeviceInterfaceDetail(deviceInfoSet,
                           &deviceInterfaceData,deviceInterfaceDetailData,
                           requiredSize,&deviceInterfaceDetailDataSize,NULL)){
                   SetupDiDestroyDeviceInfoList(deviceInfoSet);
                   releaseMemory(deviceInterfaceDetailData);
                   return 0;
                }
            }
            else {
                if(ERROR_NO_MORE_ITEMS == GetLastError()){
                  done = TRUE;
                  break;
                }
            }
            cout << deviceInterfaceDetailData->DevicePath << endl;
            releaseMemory(deviceInterfaceDetailData);
        }
    }
    SetupDiDestroyDeviceInfoList(deviceInfoSet);
    FreeLibrary(hHidLib);
    return memberIndex;
}
//-------------------------------------------------------
int main(){
   cout << "\nLiczba interfejsów urządzeń klasy HID w systemie = "\
        << searchInterfaceHidDevices() << endl;
   cout << endl;
```

```
    system("PAUSE");
    return 0;
}
//---------------------------------------------------------
```

Funkcja SetupDiGetDeviceRegistryProperty()

Zdefiniowana w module *setupapi.h* funkcja `SetupDiGetDeviceRegistryProperty()` wydobywa właściwości zainstalowanych urządzeń PnP. Właściwości te można również odczytać za pomocą edytora rejestru (patrz rozdział 2., rysunki 2.4 i 2.5).

```
WINSETUPAPI BOOL WINAPI
  SetupDiGetDeviceRegistryProperty(
    IN HDEVINFO  DeviceInfoSet,
    IN PSP_DEVINFO_DATA  DeviceInfoData,
    IN DWORD  Property,
    OUT PDWORD  PropertyRegDataType,  OPTIONAL
    OUT PBYTE  PropertyBuffer,
    IN DWORD  PropertyBufferSize,
    OUT PDWORD  RequiredSize  OPTIONAL
    );
```

Wskaźnik `DeviceInfoSet` jest zwracany przez funkcję `SetupDiGetClassDevs()`. Parametr `DeviceInfoData` wskazuje strukturę `SP_DEVINFO_DATA`. Parametr `Property` identyfikuje właściwość urządzenia PnP, którą aktualnie chcemy odczytać. Jest on reprezentowany przez odpowiednie stałe symboliczne, których podzbiór został wykorzystany w kodzie z listingu 5.3. Kompletny zestaw predefiniowanych stałych symbolicznych można odnaleźć w plikach pomocy. Opcjonalnie używany wskaźnik `PropertyRegDataType` wskazuje typ danej zawierającej właściwość urządzenia. Parametr `PropertyBuffer` wskazuje bufor danych, poprzez który odczytamy właściwości urządzenia. Jeżeli `PropertyBuffer` przypiszemy wartość `NULL`, a parametrowi `PropertyBufferSize` wartość zero, funkcja zwróci żądany rozmiar bufora danych przechowywany w zmiennej `RequiredSize`. Parametr `PropertyBufferSize` określa w bajtach rozmiar bufora wskazywanego przez `PropertyBuffer`.

Listing 5.3. *Kod modułu usb_R5_3.cpp*

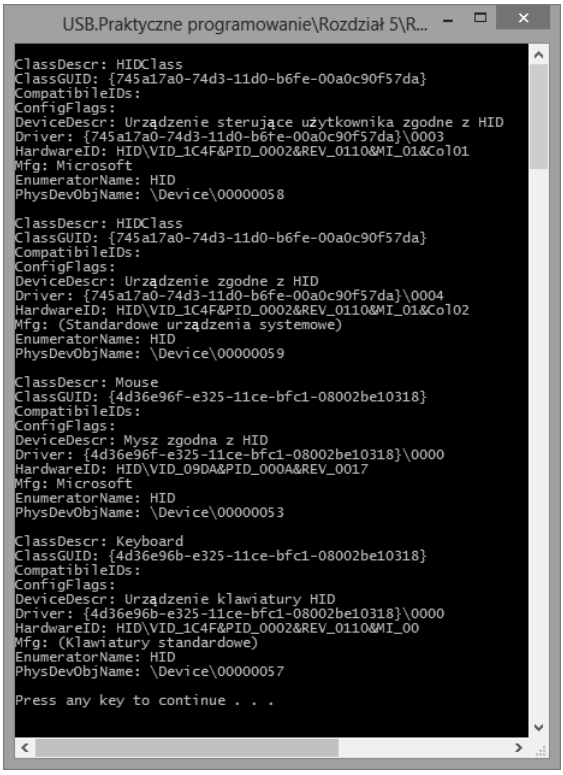
```
#include <windows>
#pragma option push -a1
#include <setupapi>
#pragma option pop
#include <assert>
#include <iostream>
#include <cstring>

using namespace std;

void displayError(const char* msg){
```

```
  cout << msg << endl;
  system("PAUSE");
  exit(0);
};
//-------------------------------------------------------
template <class T>
inline void releaseMemory(T &x)
{
  assert(x != NULL);
  delete [] x;
  x = NULL;
}
//-------------------------------------------------------
GUID classGuid;
HMODULE hHidLib;
DWORD /*unsigned long*/ memberIndex = 0;
DWORD deviceInterfaceDetailDataSize;
DWORD requiredSize;

HDEVINFO deviceInfoSet;
SP_DEVICE_INTERFACE_DATA deviceInterfaceData;
PSP_DEVICE_INTERFACE_DETAIL_DATA deviceInterfaceDetailData = NULL;
SP_DEVINFO_DATA deviceInfoData;
//-------------------------------------------------------
string getRegistryPropertyString(HDEVINFO deviceInfoSet,
                                 PSP_DEVINFO_DATA deviceInfoData, DWORD property)
{
   DWORD propertyBufferSize = 0;
   //DWORD propertyRegDataType = 0;
   char *propertyBuffer = NULL;

   SetupDiGetDeviceRegistryProperty(deviceInfoSet, deviceInfoData, property,
                                    /*&propertyRegDataType*/NULL, NULL, 0,
                                    &propertyBufferSize);

   //alokowanie pamięci dla bufora danych
   propertyBuffer = new char[(propertyBufferSize * sizeof(TCHAR))];

   bool result=SetupDiGetDeviceRegistryProperty(deviceInfoSet, deviceInfoData,
                                        property,/*&propertyRegDataType*/NULL,
                                        propertyBuffer, propertyBufferSize,
                                        NULL);
   if(!result)
      releaseMemory(propertyBuffer);
   return propertyBuffer;
}
//-------------------------------------------------------
int main(){

  void (__stdcall *HidD_GetHidGuid)(OUT LPGUID HidGuid);

  hHidLib = LoadLibrary("C:\\Windows\\system32\\HID.DLL");
  if (!hHidLib)
    displayError("Błąd dołączenia biblioteki HID.DLL.");

   (FARPROC&) HidD_GetHidGuid = GetProcAddress(hHidLib,
                                               "HidD_GetHidGuid");
   if (!HidD_GetHidGuid){
```

```
   FreeLibrary(hHidLib);
   displayError("Nie znaleziono identyfikatora GUID.");
}

HidD_GetHidGuid(&classGuid);

deviceInfoSet = SetupDiGetClassDevs(&classGuid, NULL, NULL,
                DIGCF_PRESENT | DIGCF_INTERFACEDEVICE);
if (deviceInfoSet == INVALID_HANDLE_VALUE)
   displayError("Nie zidentyfikowano podłączonych urządzeń.\n");

deviceInterfaceData.cbSize = sizeof(SP_DEVICE_INTERFACE_DATA);

while(SetupDiEnumDeviceInterfaces(deviceInfoSet, NULL, &classGuid,
                                  memberIndex, &deviceInterfaceData)){
    memberIndex++; //inkrementacja numeru interfejsu

    SetupDiGetDeviceInterfaceDetail(deviceInfoSet, &deviceInterfaceData,
                          NULL, 0, &deviceInterfaceDetailDataSize, NULL);

    deviceInterfaceDetailData = (PSP_DEVICE_INTERFACE_DETAIL_DATA)
                           new DWORD[deviceInterfaceDetailDataSize];

    deviceInterfaceDetailData->cbSize=sizeof(SP_DEVICE_INTERFACE_DETAIL_DATA);

    deviceInfoData.cbSize = sizeof(SP_DEVINFO_DATA);

    if (!SetupDiGetDeviceInterfaceDetail(deviceInfoSet, &deviceInterfaceData,
                    deviceInterfaceDetailData, deviceInterfaceDetailDataSize,
                    &requiredSize, &deviceInfoData)){
        releaseMemory(deviceInterfaceDetailData);
        SetupDiDestroyDeviceInfoList(deviceInfoSet);
        //displayError ("Nie można pobrać informacji o interfejsie.\n");
    }
   //cout << deviceInterfaceDetailData->DevicePath << endl;

    cout << "\nClassDescr: "<<getRegistryPropertyString(deviceInfoSet,
                                        &deviceInfoData, SPDRP_CLASS);
    cout << "\nClassGUID: "<<getRegistryPropertyString(deviceInfoSet,
                                        &deviceInfoData, SPDRP_CLASSGUID);
    cout << "\nCompatibileIDs: "<<getRegistryPropertyString(deviceInfoSet,
                                    &deviceInfoData, SPDRP_COMPATIBLEIDS);
    cout << "\nConfigFlags: "<<getRegistryPropertyString(deviceInfoSet,
                                    &deviceInfoData, SPDRP_CONFIGFLAGS);
    cout << "\nDeviceDescr: "<<getRegistryPropertyString(deviceInfoSet,
                                    &deviceInfoData, SPDRP_DEVICEDESC);
    cout << "\nDriver: "<<getRegistryPropertyString(deviceInfoSet,
                                    &deviceInfoData, SPDRP_DRIVER);
    //cout << "\nFriendlyName: "<<getRegistryPropertyString(deviceInfoSet,
    //                                  &deviceInfoData, SPDRP_FRIENDLYNAME);
    cout << "\nHardwareID: "<<getRegistryPropertyString(deviceInfoSet,
                                    &deviceInfoData, SPDRP_HARDWAREID);
    cout << "\nMfg: "<<getRegistryPropertyString(deviceInfoSet,
                                    &deviceInfoData, SPDRP_MFG);
    cout << "\nEnumeratorName: "<<getRegistryPropertyString(deviceInfoSet,
                                    &deviceInfoData, SPDRP_ENUMERATOR_NAME);
    cout << "\nPhysDevObjName: "<<getRegistryPropertyString(deviceInfoSet,
                          &deviceInfoData, SPDRP_PHYSICAL_DEVICE_OBJECT_NAME);
```

```
            cout << endl;
            releaseMemory(deviceInterfaceDetailData);
        }; //koniec while

        SetupDiDestroyDeviceInfoList(deviceInfoSet);
        FreeLibrary(hHidLib);
        cout << endl;
        system("PAUSE");
        return 0;
    }
    //-------------------------------------------------------------
```

Używanie funkcji `SetupDiGetDeviceRegistryProperty()` z reguły nie ogranicza się do pojedynczego wywołania. W pierwszym wywołaniu określamy wymagany rozmiar bufora danych, a w następnym odczytujemy żądane właściwości zainstalowanego w systemie urządzenia PnP. Pokazano to na listingu 5.3 w ciele przykładowej funkcji `getRegistryPropertyString()` wydobywającej niektóre właściwości łańcuchowe sprzętu zgodnego z klasą HID. Rysunek 5.7 przedstawia wynik działania programu cyklicznie wywołującego funkcję `getRegistryPropertyString()` w celu odczytania wybranych właściwości łańcuchowych urządzeń zainstalowanych w systemie.



Rysunek 5.7. *Aplikacja proj_USB_R5_3 w trakcie działania*

W rejestrze systemowym elementami podkluczy tematycznych są dwa typy danych: łańcuchowe (oznaczone jako `REG_SZ` lub `REG_MULTI_SZ`) i liczbowe (oznaczone jako `REG_DWORD`).

Testując kod z listingu 5.3, możemy zauważyć, że funkcja `getRegistryPropertyString()` wydobywa jedynie właściwości łańcuchowe urządzenia. Aby odzyskać właściwość liczbową, należy zmodyfikować tę funkcję lub, co jest dużo bardziej pożyteczne, zbudować jej odmianę, która będzie zwracała dane typu `DWORD`. Na listingu 5.4 zamieszczono odpowiedni przykład funkcji `getRegistryPropertyDWORD()`, wydobywającej właściwości liczbowe urządzenia zapisane w rejestrze systemowym.

Listing 5.4. *Kod funkcji getRegistryPropertyDWORD() wraz z jej przykładowym wywołaniem*

```
//...
SP_DEVINFO_DATA deviceInfoData;
//-------------------------------------------------------
DWORD getRegistryPropertyDWORD(HDEVINFO deviceInfoSet,
                          PSP_DEVINFO_DATA deviceInfoData, DWORD property)
{
   DWORD propertyBufferSize = 0;
   DWORD propertyRegDataType = 0;
   DWORD *propertyBuffer = 0;
   bool result;

   SetupDiGetDeviceRegistryProperty(deviceInfoSet, deviceInfoData, property,
                                    &propertyRegDataType, NULL, 0,
                                    &propertyBufferSize);

   //alokowanie pamięci dla bufora danych
   propertyBuffer = new DWORD[(propertyBufferSize * sizeof(DWORD))];

   result = SetupDiGetDeviceRegistryProperty(deviceInfoSet, deviceInfoData,
                                   property,&propertyRegDataType,
                                   (char*)propertyBuffer, sizeof(propertyBuffer),
                                   NULL);
   if(!result)
      releaseMemory(propertyBuffer);
   return *propertyBuffer;
}
//-------------------------------------------------------
int main(){
//...
   cout<<"\nCapabilities" << getRegistryPropertyDWORD(deviceInfoSet,
                            &deviceInfoData, SPDRP_CAPABILITIES);
//...
}
//-------------------------------------------------------
```

Kup książkę Poleć książkę

Struktury danych

Elementy opisu urządzeń USB, które są dostępne w systemie, są często przechowywane w polach odpowiednio skonstruowanych struktur danych znajdujących się w jednej przestrzeni nazw. Na listingu 5.4 pokazano przykładową strukturę DEVICE_DATA:

```
typedef struct _DEVICE_DATA {
  TCHAR  *HardwareId; //identyfikator sprzętu
  TCHAR  *Path; //łącze symboliczne
  DWORD  DeviceInstance;
} DEVICE_DATA, *PDEVICE_DATA;
//-------------------------------------------------------
```

Jej pola przechowują przykładowe dane zawierające identyfikator sprzętu (HardwareId), ścieżkę dostępu do interfejsu urządzenia (Path) oraz DeviceInstance, który będzie lokalizował element DevInst struktury USB_DEFINFO_DATA. Zawartość struktury DEVICE_DATA pozwala wstępnie zidentyfikować urządzenie.

Aby w pełni wykorzystać tak skonstruowany typ danych, należy zadeklarować tablicę wskaźników do struktur o rozmiarze nie mniejszym niż rzeczywista liczba interfejsów urządzeń USB w systemie:

```
PDEVICE_DATA deviceList;
//...
deviceList = (PDEVICE_DATA)new \
             DEVICE_DATA[((memberIndex+1)*sizeof(DEVICE_DATA))];
```

Warto pamiętać, że w przypadku błędnego zaalokowania pamięci dla tablicy struktur kompilator zgłosi błąd naruszenia pamięci.

Po prawidłowym zaalokowaniu tablicy struktur określamy za pomocą funkcji strlen() aktualną długość ścieżki dostępu do interfejsu urządzenia:

```
size_t nLen = strlen(deviceInterfaceDetailData->DevicePath) + 1;
```

oraz tworzymy tablicę ścieżek:

```
deviceList[memberIndex].Path = new TCHAR[(nLen*sizeof(TCHAR))];
```

Ścieżka dostępu do interfejsu urządzenia zostaje przekopiowana za pomocą funkcji strncpy() do pola Path elementu (o indeksie memberIndex) tablicy struktur DEVICE_DATA.

```
strncpy(deviceList[memberIndex].Path,
                  deviceInterfaceDetailData->DevicePath, nLen);
```

Od tego momentu program dysponuje indeksowaną ścieżką dostępu do interfejsu urządzenia, którą można wykorzystać na przykład w funkcji uzyskującej dostęp do pliku sterownika urządzenia:

```
CreateFile(deviceList[2].Path, ...);
```

Sposoby wypełniania pozostałych elementów struktury DEVICE_DATA zaprezentowano w kodzie z listingu 5.5, zawierającym funkcję setGetHidDeviceData(). Na rysunku 5.8 pokazano wynik działania omawianego programu.

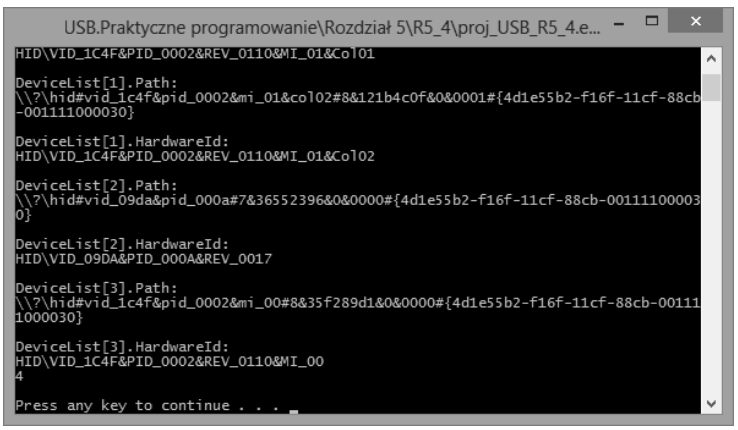
Kup książkę Poleć książkę

Listing 5.5. *Kod modułu usb_R5_4.cpp*

```
#include <windows>
#pragma option push -a1
  #include <setupapi>
#pragma option pop
#include <assert>
#include <iostream>
#include <cstring>

using namespace std;

void displayError(const char* msg){
  cout << msg << endl;
  system("PAUSE");
  exit(0);
};
//---------------------------------------------------------
template <class T>
inline void releaseMemory(T &x)
{
  assert(x != NULL);
  delete [] x;
  x = NULL;
}
//---------------------------------------------------------
typedef struct _DEVICE_DATA {
  TCHAR  *HardwareId;
  TCHAR  *Path; //łącze symboliczne
  DWORD  DeviceInstance;
} DEVICE_DATA, *PDEVICE_DATA;
//---------------------------------------------------------
int setGetHidDeviceData()
{
    PDEVICE_DATA deviceList;
    DWORD propertyBufferSize = 0;
    char *propertyBuffer = NULL;
    SP_DEVINFO_DATA deviceInfoData;

    HMODULE hHidLib;
    HDEVINFO deviceInfoSet;
    SP_INTERFACE_DEVICE_DATA deviceInterfaceData;
    DWORD  memberIndex = 0;
    GUID classGuid;
    PSP_DEVICE_INTERFACE_DETAIL_DATA deviceInterfaceDetailData = NULL;
    DWORD requiredSize = 0;
    DWORD deviceInterfaceDetailDataSize = 0;
    DWORD searchMaxDevice = 100; //maksymalna liczba interfejsów urządzeń
    bool done = false;

    void (__stdcall *HidD_GetHidGuid)(OUT LPGUID HidGuid);

    hHidLib = LoadLibrary("C:\\Windows\\system32\\HID.DLL");
    if (!hHidLib)
      displayError("Błąd dołączenia biblioteki HID.DLL.");

    (FARPROC&) HidD_GetHidGuid = GetProcAddress(hHidLib,
                                          "HidD_GetHidGuid");
```

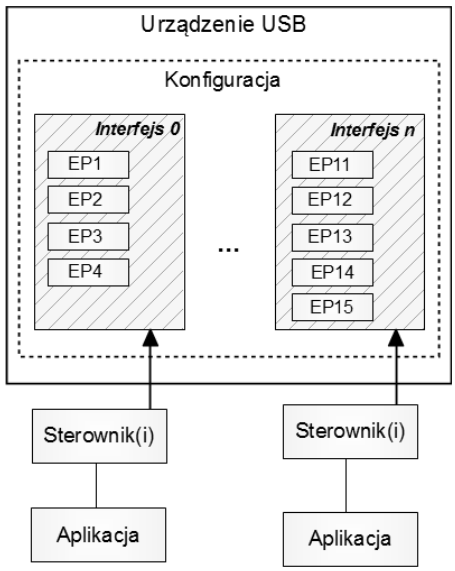
```
if (!HidD_GetHidGuid){
   FreeLibrary(hHidLib);
   displayError("Nie znaleziono identyfikatora GUID.");
}

HidD_GetHidGuid (&classGuid);

deviceInfoSet = SetupDiGetClassDevs(&classGuid, NULL, NULL,
                                    (DIGCF_PRESENT | DIGCF_DEVICEINTERFACE));

deviceInterfaceData.cbSize = sizeof(SP_INTERFACE_DEVICE_DATA);

while(!done) {
   deviceList = new DEVICE_DATA[((memberIndex+1)*sizeof(DEVICE_DATA))];

   for(; memberIndex < searchMaxDevice; memberIndex++) {
      if(SetupDiEnumDeviceInterfaces(deviceInfoSet,0,&classGuid,
                                     memberIndex,&deviceInterfaceData)) {
        SetupDiGetDeviceInterfaceDetail(deviceInfoSet,&deviceInterfaceData,
                                     NULL,0,&deviceInterfaceDetailDataSize,
                                     NULL);
        requiredSize = deviceInterfaceDetailDataSize;
        deviceInterfaceDetailData = (PSP_DEVICE_INTERFACE_DETAIL_DATA)\
                          new DWORD[deviceInterfaceDetailDataSize];

           if(deviceInterfaceDetailData) {
              deviceInterfaceDetailData->cbSize=\
                          sizeof(SP_INTERFACE_DEVICE_DETAIL_DATA);
            }
            else {
               SetupDiDestroyDeviceInfoList(deviceInfoSet);
               releaseMemory(deviceInterfaceDetailData);
               return 0;
            }
            deviceInfoData.cbSize = sizeof(SP_DEVINFO_DATA);
            if (!SetupDiGetDeviceInterfaceDetail(deviceInfoSet,
                        &deviceInterfaceData, deviceInterfaceDetailData,
                        deviceInterfaceDetailDataSize,
                        &requiredSize, &deviceInfoData)) {
               SetupDiDestroyDeviceInfoList(deviceInfoSet);
               releaseMemory(deviceInterfaceDetailData);
               return 0;
            }

        size_t nLen = strlen(deviceInterfaceDetailData->DevicePath) + 1;
        deviceList[memberIndex].Path = new TCHAR[(nLen*sizeof(TCHAR))];

        strncpy(deviceList[memberIndex].Path,
                          deviceInterfaceDetailData->DevicePath, nLen);

        cout <<"\nDeviceList["<<memberIndex<<"].Path: \n" <<
                deviceList[memberIndex].Path << endl;

        deviceList[memberIndex].DeviceInstance = deviceInfoData.DevInst;

        SetupDiGetDeviceRegistryProperty(deviceInfoSet, &deviceInfoData,
                                     SPDRP_HARDWAREID, NULL, NULL, 0,
                                     &propertyBufferSize);
```

```
            //alokowanie pamięci dla bufora danych
            propertyBuffer = new char[(propertyBufferSize*sizeof(TCHAR))];

            SetupDiGetDeviceRegistryProperty(deviceInfoSet, &deviceInfoData,
                                             SPDRP_HARDWAREID,NULL,
                                             propertyBuffer, propertyBufferSize,
                                             NULL);

            deviceList[memberIndex].HardwareId  = propertyBuffer;
            cout <<"\nDeviceList["<<memberIndex<<"].HardwareId: \n" <<
                     deviceList[memberIndex].HardwareId << endl;

            }
            else {
                if(ERROR_NO_MORE_ITEMS == GetLastError()){
                  done = TRUE;
                  break;
                }
            }
            releaseMemory(propertyBuffer);
            releaseMemory(deviceInterfaceDetailData);
            releaseMemory(deviceList[memberIndex].Path);
        }
        releaseMemory(deviceList);
    }
    SetupDiDestroyDeviceInfoList(deviceInfoSet);
    FreeLibrary(hHidLib);
    return memberIndex;
}
//-------------------------------------------------------------
int main(){
   cout << setGetHidDeviceData() << endl;
   cout << endl;
   system("PAUSE");
   return 0;
}
//-------------------------------------------------------------
```



Rysunek 5.8. *Aplikacja proj_USB_R5_4 w trakcie działania*

Na rysunku 5.9 zaprezentowano sposób uzyskiwania dostępu do urządzenia USB. W pierwszej kolejności należy określić ścieżkę dostępu do właściwego dla danej konfiguracji interfejsu, jaki sterownik udostępnia warstwie aplikacji, po czym uzyskać identyfikator pliku sterownika urządzenia.



Rysunek 5.9. *Uzyskiwanie dostępu do złożonego urządzenia USB funkcjonującego w podstawowym modelu konfiguracji (por. rysunek 3.13)*

Program użytkownika uzyskuje dostęp do urządzenia poprzez jego interfejs dostarczany przez obiekty PDO i FDO. Dla każdego zidentyfikowanego przez system portu USB oraz dla każdego zidentyfikowanego i przyłączonego urządzenia określony sterownik tworzy odpowiedni obiekt urządzenia, interfejs, wewnętrzną nazwę obiektu fizycznego (`PhysDevObjName` — patrz rysunek 5.7) oraz nazwę reprezentującą łącze symboliczne do pliku sterownika (patrz rysunek 5.8).

Gdy programista zna numer oraz identyfikator `GUID` żądanego interfejsu urządzenia, może wykorzystać pokazaną na listingu 5.6 funkcję wydobywającą pełną ścieżkę dostępu do interfejsu, jaki sterownik urządzenia udostępnia warstwie aplikacji.

Listing 5.6. *Jeden ze sposobów uzyskiwania pełnej ścieżki wystąpienia obiektu urządzenia na podstawie znajomości numeru oraz identyfikatora GUID interfejsu urządzenia*

```
#include <initguid>
//...
//-------------------------------------------------------
DEFINE_GUID(devInterfaceGUIDConstant, 0x4d1e55b2, 0xf16f, 0x11cf, \
                        0x88, 0xcb, 0x00, 0x11, 0x11, 0x00, 0x00, 0x30);
GUID devInterfaceGUID = devInterfaceGUIDConstant;
//-------------------------------------------------------
char* getDevicePath(LPGUID devInterfaceGUID, /*UINT interfaceIndex*/)
{
   HDEVINFO deviceInfoSet;
   SP_DEVICE_INTERFACE_DATA deviceInterfaceData;
   PSP_DEVICE_INTERFACE_DETAIL_DATA deviceInterfaceDetailData = NULL;
```

```
    DWORD requiredSize, deviceInterfaceDetailDataSize = 0;
    BOOL bResult;

    deviceInfoSet = SetupDiGetClassDevs(devInterfaceGUID,
                                        NULL, NULL,
                                        (DIGCF_PRESENT |
                                        DIGCF_DEVICEINTERFACE));
    if(deviceInfoSet == INVALID_HANDLE_VALUE) {
      //błąd
      exit(1);
    }
    deviceInterfaceData.cbSize = sizeof(SP_DEVICE_INTERFACE_DATA);
    bResult = SetupDiEnumDeviceInterfaces(deviceInfoSet, NULL,
                                          devInterfaceGUID,
                                          1, /*interfaceIndex numer interfejsu*/
                                          &deviceInterfaceData);
    if(bResult == FALSE) {
      //błąd
      SetupDiDestroyDeviceInfoList(deviceInfoSet);
      exit(1);
    }
    SetupDiGetDeviceInterfaceDetail(deviceInfoSet,&deviceInterfaceData,
                                    NULL,0,&deviceInterfaceDetailDataSize,
                                    NULL);
    deviceInterfaceDetailData = (PSP_DEVICE_INTERFACE_DETAIL_DATA)
                              new DWORD[deviceInterfaceDetailDataSize];

    if(deviceInterfaceDetailData == NULL) {
       SetupDiDestroyDeviceInfoList(deviceInfoSet);
       //błąd alokacji pamięci
       exit(1);
    }
    deviceInterfaceDetailData->cbSize = sizeof(SP_DEVICE_INTERFACE_DETAIL_DATA);
    bResult = SetupDiGetDeviceInterfaceDetail(deviceInfoSet,
                                              &deviceInterfaceData,
                                              deviceInterfaceDetailData,
                                              deviceInterfaceDetailDataSize,
                                              &requiredSize,NULL);

    if(bResult == FALSE) {
       //błąd
       SetupDiDestroyDeviceInfoList(deviceInfoSet);
       delete [] deviceInterfaceDetailData;
       exit(1);
    }
    return deviceInterfaceDetailData->DevicePath;
}
//-------------------------------------------------------
int main()
{
  cout << getDevicePath(&devInterfaceGUID, /*...*/);
  devObject = CreateFile(getDevicePath(&devInterfaceGUID),/*...*/);
  //Patrz rozdział 6.
  //...
}
//-------------------------------------------------------
```

Moduł usbiodef.h

Dotychczas zostały omówione procedury detekcji i identyfikacji urządzeń klasy HID aktualnie podłączonych do magistrali USB. Warto pamiętać, że w zasobach WDK można odszukać użyteczny moduł *usbiodef.h*, w którym m.in. zdefiniowanych jest wiele identyfikatorów `GUID`, za pomocą których uzyskuje się ścieżki dostępu do interfejsów wszystkich urządzeń USB aktualnie dostępnych w systemie.

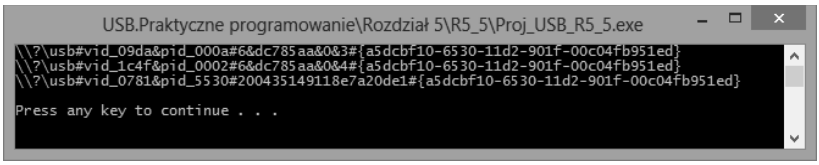
Jeżeli w przypadku urządzeń HID zdecydujemy się posługiwać identyfikatorem `GUID_DEVINTERFACE_HID`, w kodzie programu należy zrezygnować z funkcji `HidD_GetHidGuid()`, tak jak pokazano na listingu 5.7. Na rysunku 5.10 zaprezentowano program w trakcie działania.

Listing 5.7. *Kod modułu usb_R5_5.cpp*

```
#include <windows>
#pragma option push -a1
  #include <setupapi>
#pragma option pop
#include <assert>
#include <iostream>

using namespace std;

void displayError(const char* msg){
  cout << msg << endl;
  system("PAUSE");
  exit(0);
};
//---------------------------------------------------------
template <class T>
inline void releaseMemory(T &x)
{
  assert(x != NULL);
  delete [] x;
  x = NULL;
}
//---------------------------------------------------------
/*A5DCBF10-6530-11D2-901F-00C04FB951ED */
static GUID GUID_DEVINTERFACE_USB_DEVICE =
{0xA5DCBF10, 0x6530, 0x11D2, {0x90, 0x1F, 0x00, 0xC0,
                              0x4F, 0xB9, 0x51, 0xED}};

/*3ABF6F2D-71C4-462a-8A92-1E6861E6AF27*/
static GUID GUID_DEVINTERFACE_USB_HOST_CONTROLLER =
{0x3abf6f2d, 0x71c4, 0x462a, {0x8a, 0x92, 0x1e, \
                              0x68, 0x61, 0xe6, 0xaf, 0x27}};

/*F18A0E88-C30C-11D0-8815-00A0C906BED8*/
static GUID GUID_DEVINTERFACE_USB_HUB =
{0xf18a0e88, 0xc30c, 0x11d0,{0x88, 0x15, 0x00, \
                             0xa0, 0xc9, 0x06, 0xbe, 0xd8}};
```

Kup książkę Poleć książkę

```
/*4D1E55B2-F16F-11CF-88CB-001111000030*/
static GUID GUID_DEVINTERFACE_HID =
{0x4D1E55B2, 0xF16F, 0x11CF, {0x88, 0xCB, 0x00, \
                              0x11, 0x11, 0x00, 0x00, 0x30}};

//-------------------------------------------------------
DWORD memberIndex = 0;
DWORD deviceInterfaceDetailDataSize;
DWORD requiredSize;

HDEVINFO deviceInfoSet;
SP_DEVICE_INTERFACE_DATA deviceInterfaceData;
PSP_DEVICE_INTERFACE_DETAIL_DATA deviceInterfaceDetailData = NULL;

int main(){

   deviceInfoSet = SetupDiGetClassDevs(&GUID_DEVINTERFACE_USB_DEVICE,
                   NULL, NULL, DIGCF_PRESENT | DIGCF_INTERFACEDEVICE);
   if (deviceInfoSet == INVALID_HANDLE_VALUE)
      displayError("Nie zidentyfikowano podłączonych urządzeń.\n");

   deviceInterfaceData.cbSize = sizeof(SP_DEVICE_INTERFACE_DATA);

   while(SetupDiEnumDeviceInterfaces(deviceInfoSet, NULL,
                                     &GUID_DEVINTERFACE_USB_DEVICE,
                                     memberIndex, &deviceInterfaceData)){
       memberIndex++; //inkrementacja numeru interfejsu

       SetupDiGetDeviceInterfaceDetail(deviceInfoSet, &deviceInterfaceData,
                             NULL, 0, &deviceInterfaceDetailDataSize, NULL);

       deviceInterfaceDetailData = (PSP_DEVICE_INTERFACE_DETAIL_DATA)
                              new DWORD[deviceInterfaceDetailDataSize];

       deviceInterfaceDetailData->cbSize=sizeof(SP_DEVICE_INTERFACE_DETAIL_DATA);

       if (!SetupDiGetDeviceInterfaceDetail(deviceInfoSet, &deviceInterfaceData,
                       deviceInterfaceDetailData, deviceInterfaceDetailDataSize,
                       &requiredSize, NULL)){
           releaseMemory(deviceInterfaceDetailData);
           SetupDiDestroyDeviceInfoList(deviceInfoSet);
           //displayError ("Nie można pobrać informacji o interfejsie.\n");
       }

       cout << deviceInterfaceDetailData->DevicePath << endl;
       releaseMemory(deviceInterfaceDetailData);
    };//koniec while

   SetupDiDestroyDeviceInfoList(deviceInfoSet);
   cout << endl;
   system("PAUSE");
   return 0;
}
//-------------------------------------------------------
```



Rysunek 5.10. *Detekcja wszystkich urządzeń aktualnie podłączonych do magistrali USB z GUID_DEVINTERFACE_USB_DEVICE*

Moduł cfgmgr32.h

Menedżer konfiguracji oferuje szereg niezwykle użytecznych i prostych w wykorzystaniu funkcji rodziny `CM_xxx()`, które są pomocne w szybkim zdiagnozowaniu aktualnie dostępnych w systemie urządzeń. W niniejszym podrozdziale zostaną przedstawione te najbardziej podstawowe. W celu wykorzystania zasobów menedżera do identyfikacji urządzeń USB w pierwszej kolejności należy określić interesującą nas klasę instalacji urządzeń w funkcji `SetupDiGetClassDevs()` oraz odpowiednio wywołać żądaną funkcję menedżera. Na listingu 5.8 zaprezentowano przykład wykorzystania funkcji:

```
CMAPI CONFIGRET
WINAPI CM_Get_DevNode_Registry_Property(
  IN     DEVINST dnDevInst,
  IN     ULONG ulProperty,
  INOUT  PULONG pulRegDataType,
  INOUT  PVOID Buffer,
  INOUT  PULONG pulLength,
  IN     ULONG ulFlags
);
```

w celu szybkiego odczytania informacji o zainstalowanych i aktualnie dostępnych urządzeniach na podstawie zapisów rejestru systemowego. Pierwszym parametrem funkcji jest pole typu `DEVINST`. `DEVINST` jest wewnętrzną strukturą danych reprezentującą urządzenie w systemie. Dla urządzeń USB pierwszym parametrem wejściowym funkcji `CM_Get_DevNode_Registry_Property()` będzie pole `DevInst` struktury `SP_DEVINFO_DATA` przechowującej informacje na temat egzemplarza urządzenia należącego do klasy urządzeń USB. Parametr wejściowy `ulProperty` identyfikuje właściwość urządzenia, którą aktualnie chcemy odczytać. Jest on reprezentowany przez odpowiednie stałe symboliczne `CM_DRP_xxx`, określające żądane właściwości instalacyjne urządzenia zapisane w rejestrze systemowym (patrz rozdział 2.). Wartości `CM_DRP_xxx` są zdefiniowane w module *cfgmgr32.h* w następujący sposób:

```
CM_DRP_DEVICEDESC                (0x00000001) //DeviceDesc REG_SZ property(RW)
CM_DRP_HARDWAREID                (0x00000002) //HardwareID REG_MULTI_SZ
                                              //property(RW)
CM_DRP_COMPATIBLEIDS             (0x00000003) //CompatibleIDs REG_MULTI_SZ
                                              //property(RW)
CM_DRP_UNUSED0                   (0x00000004) //unused
CM_DRP_SERVICE                   (0x00000005) //Service REG_SZ property(RW)
CM_DRP_UNUSED1                   (0x00000006) //unused
CM_DRP_UNUSED2                   (0x00000007) //unused
CM_DRP_CLASS                     (0x00000008) //Class REG_SZ property(RW)
```

Kup książkę Poleć książkę

```
CM_DRP_CLASSGUID                  (0x00000009) //ClassGUID REG_SZ property(RW)
CM_DRP_DRIVER                     (0x0000000A) //Driver REG_SZ property(RW)
CM_DRP_CONFIGFLAGS                (0x0000000B) //ConfigFlags REG_DWORD
                                               //property(RW)
CM_DRP_MFG                        (0x0000000C) //Mfg REG_SZ property(RW)
CM_DRP_FRIENDLYNAME               (0x0000000D) //FriendlyName REG_SZ property(RW)
CM_DRP_LOCATION_INFORMATION       (0x0000000E) //LocationInformation REG_SZ
                                               //property(RW)
CM_DRP_PHYSICAL_DEVICE_OBJECT_NAME (0x0000000F) //PhysicalDeviceObjectName REG_SZ
                                               //property(R)
CM_DRP_CAPABILITIES               (0x00000010) //Capabilities REG_DWORD
                                               //property(R)
CM_DRP_UI_NUMBER                  (0x00000011) //UiNumber REG_DWORD property(R)
CM_DRP_UPPERFILTERS               (0x00000012) //UpperFilters REG_MULTI_SZ
                                               //property(RW)
```

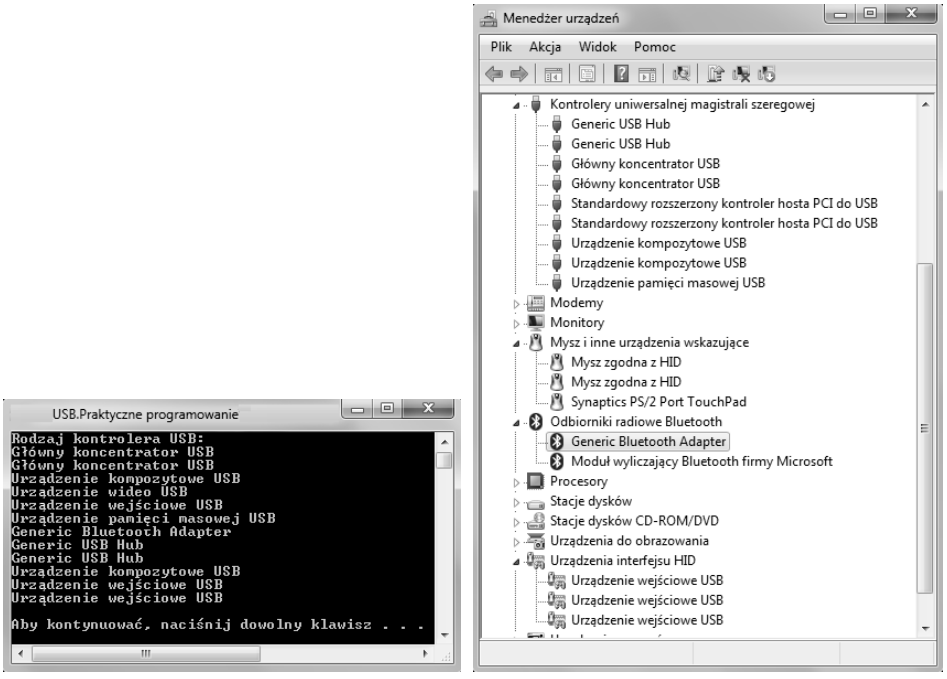
Opcjonalny wskaźnik `pulRegDataType` wskazuje wartość `NULL`, a wskaźnik `Buffer` — bufor danych przechowujący łańcuch znaków identyfikujący egzemplarz urządzenia aktualnie przyłączonego do magistrali USB. Wskaźnik `pulLength` określa długość bufora danych. Znacznik `ulFlags` nie ma istotnego znaczenia i może zawierać wartość `NULL`. Prawidłowo wykonane funkcje menedżera konfiguracji zwracają wartość `CR_SUCCESS`. W programach x64 zamiast `CM_Get_DevNode_Registry_Property()` powinno się używać funkcji `SetupDiGetDeviceRegistryProperty()`, tak jak pokazano w dalszej części niniejszego podrozdziału.

Listing 5.8. *Określenie typów urządzeń USB aktualnie zainstalowanych w systemie*

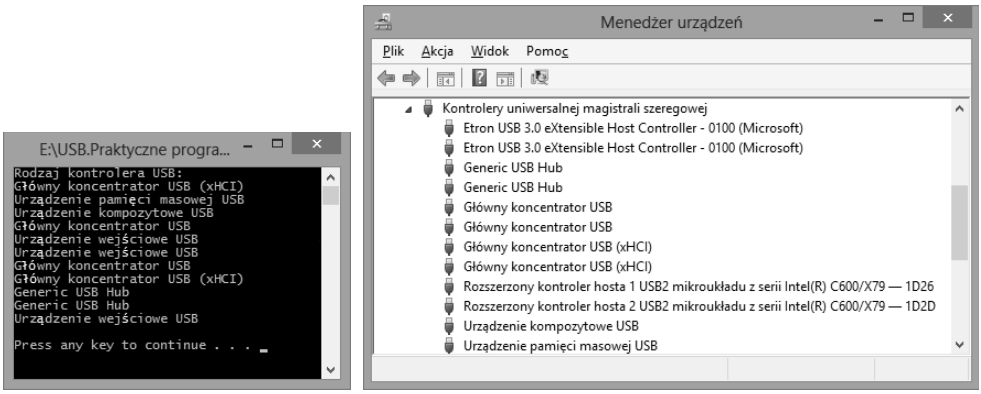
```
#include <windows>
#pragma option push -a1
   #include <setupapi>
#pragma option pop
#include <iostream>
#include "D:\\WINDDK\\7600.16385.1\\inc\\api\\cfgmgr32.h"

using namespace std;
//-------------------------------------------------------------
//Wyświetla listę urządzeń USB zainstalowanych w systemie
string getDeviceDescription(DEVINST devInst)
{
  char /*TCHAR*/ buffer[1023];
  ULONG bufferLen;
  bufferLen = sizeof(buffer);
  if ((devInst != 0) && (CM_Get_DevNode_Registry_Property(devInst,
       CM_DRP_DEVICEDESC /*CM_DRP_CLASS*/, NULL, buffer, &bufferLen, 0)
       == CR_SUCCESS))
    return buffer;
};
//-------------------------------------------------------------
void printUSBDevices()
{
  DWORD memberIndex =0;
  HDEVINFO deviceInfoSet;
  SP_DEVINFO_DATA deviceInfoData;

  deviceInfoSet = SetupDiGetClassDevs (NULL, "USB", NULL,
                  DIGCF_PRESENT|DIGCF_ALLCLASSES);
```

```
    if (deviceInfoSet == INVALID_HANDLE_VALUE)
        return;

    for(memberIndex = 0; ; memberIndex++)  {
       deviceInfoData.cbSize = sizeof (deviceInfoData);
       if(!SetupDiEnumDeviceInfo(deviceInfoSet, memberIndex,
                                 &deviceInfoData))
          break;
       cout << getDeviceDescription(deviceInfoData.DevInst) << endl;
     }
     return;
}
//-------------------------------------------------------
int main()
{
   cout << "Rodzaj kontrolera USB: \n";
   printUSBDevices();
   cout << endl;
   system("PAUSE");
   return 0;
}
//-------------------------------------------------------
```

Na rysunkach 5.11 oraz 5.12 zaprezentowano wynik działania programu z listingu 5.8 odpowiednio w systemach Windows 7 z USB 2.0 oraz Windows 8 z USB 3.0.



Rysunek 5.11. *Porównanie działania aplikacji projektu proj_USB_R5_6 z informacjami przechowywanymi w menedżerze urządzeń Windows 7 z USB 2.0*



Rysunek 5.12. *Porównanie działania aplikacji projektu proj_USB_R5_6 z informacjami przechowywanymi w menedżerze urządzeń Windows 8 z USB 3.0*

W trakcie tworzenia oprogramowania dla urządzeń USB często zachodzi potrzeba szybkiego odtworzenia drzewa urządzeń zainstalowanych w systemie oraz odczytania odpowiednich identyfikatorów VID oraz PID. Czynność tę można wykonać bez znajomości właściwych identyfikatorów `GUID` klas urządzeń. Menedżer konfiguracji oferuje funkcję:

```
CMAPI CONFIGRET WINAPI
CM_Get_Device_ID(
  IN  DEVINST dnDevInst,
  OUT PWSTR Buffer,
  IN  ULONG BufferLen,
  OUT ULONG ulFlags
);
```

przechowującą identyfikatory egzemplarzy aktualnie zainstalowanych w systemie urządzeń. Dla urządzeń USB pierwszym parametrem funkcji będzie pole `DevInst` struktury `SP_DEVINFO_DATA` przechowującej informacje na temat egzemplarza urządzenia należącego do klasy urządzeń USB. Wskaźnik `Buffer` wskazuje bufor danych zawierający łańcuch znaków opisujący egzemplarz urządzenia aktualnie przyłączonego do magistrali USB. Parametr `BufferLength` określa długość bufora danych. Znacznik `ulFlags` nie ma istotnego znaczenia i może zawierać wartość `NULL`, tak jak pokazano na listingu 5.9. Programy użytkownika mogą dodatkowo korzystać z usług funkcji:

```
CMAPI CONFIGRET WINAPI
CM_Get_Child(
   OUT  PDEVINST pdnDevInst,
   IN   DEVINST dnDevInst ,
   IN   ULONG ulFlags
);
```

przechowującej identyfikator urządzenia potomnego w drzewie urządzeń.

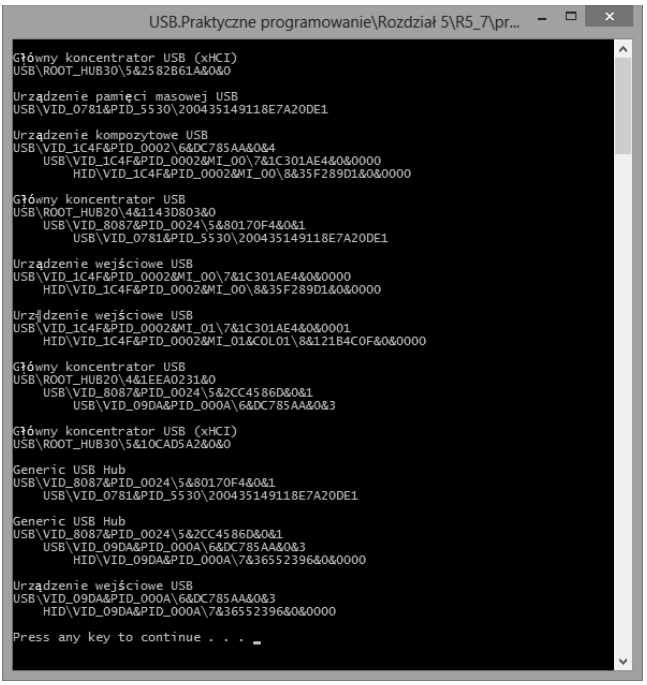
Listing 5.9. *Kod modułu usb_R5_7.cpp*

```
#include <windows>
#pragma option push -a1
   #include <setupapi>
#pragma option pop
```

```
#include <iostream>
#include "D:\\WINDDK\\7600.16385.1\\inc\\api\\cfgmgr32.h"

using namespace std;
//-------------------------------------------------------
HDEVINFO deviceInfoSet;
DEVINST devInstChild;
SP_DEVINFO_DATA deviceInfoData;
CONFIGRET configRet;
char /*TCHAR*/ buffer[MAX_DEVICE_ID_LEN];
DWORD /*ULONG*/ propertyBufferSize = 0;
DWORD property;
char *propertyBuffer = NULL;
DWORD propertyRegDataType;
//-------------------------------------------------------
char* printPrperty()
{
   SetupDiGetDeviceRegistryProperty(deviceInfoSet, &deviceInfoData,
                                    property, NULL, NULL, 0,
                                    &propertyBufferSize);
   propertyBuffer = new char[(propertyBufferSize * sizeof(char/*TCHAR*/))];
   if (SetupDiGetDeviceRegistryProperty(deviceInfoSet, &deviceInfoData,
                                        SPDRP_DEVICEDESC,
                                        &propertyRegDataType,
                                        propertyBuffer,
                                        propertyBufferSize,
                                        &propertyBufferSize))
   return propertyBuffer;
}
//---Wyświetla drzewo urządzeń USB oraz odczytuje ich identyfikatory VID i PID---
void printUSBDevices()
{
  deviceInfoSet = SetupDiGetClassDevs(NULL, "USB", NULL,
                                      DIGCF_PRESENT | DIGCF_ALLCLASSES);
  if(deviceInfoSet == INVALID_HANDLE_VALUE)
     return;

  for(DWORD memberIndex = 0; ; memberIndex++)  {
     deviceInfoData.cbSize = sizeof (deviceInfoData);
     if(!SetupDiEnumDeviceInfo(deviceInfoSet, memberIndex,
                               &deviceInfoData))
       break;

     configRet = CM_Get_Device_ID(deviceInfoData.DevInst, buffer, MAX_PATH, 0);
     if (configRet == CR_SUCCESS) {
          printf("\n%s\n", printPrperty());
          printf("%s\n", buffer);
          delete [] propertyBuffer;
     }
     configRet = CM_Get_Child(&devInstChild, deviceInfoData.DevInst, 0);
     if(configRet == CR_SUCCESS) {
         configRet = CM_Get_Device_ID (devInstChild, buffer, MAX_PATH, 0);
         if(configRet == CR_SUCCESS){
            //printf("    %s\n", printProperty());
            printf("    %s\n", buffer);
            //delete [] propertyBuffer;
         }
         configRet = CM_Get_Child(&devInstChild, devInstChild, 0);
```

```
            if(configRet == CR_SUCCESS) {
                configRet = CM_Get_Device_ID(devInstChild, buffer, MAX_PATH, 0);
                if(configRet == CR_SUCCESS) {
                   //printf("        %s\n", printProperty());
                   printf("        %s\n", buffer);
                   //delete [] propertyBuffer;
                }
            }
            else {
               continue;
            }
        }
        else {
           continue;
        }
    }
    return;
}
//-------------------------------------------------------------
int main()
{
    printUSBDevices();
    cout << endl;
    system("PAUSE");
    return 0;
}
//-------------------------------------------------------------
```

Na rysunku 5.13 zaprezentowano rezultat działania programu wyświetlającego drzewo aktualnie dostępnych w systemie urządzeń USB.



Rysunek 5.13. *Detekcja ścieżek wystąpień wszystkich obiektów urządzeń aktualnie podłączonych do magistrali USB*

Zasoby menedżera konfiguracji umożliwiają samodzielne odtworzenie drzewa urządzeń. Hierarchiczne drzewo urządzeń zawiera dane o dostępnym w systemie sprzęcie i zostaje utworzone przez system operacyjny na podstawie informacji otrzymanych z poszczególnych sterowników. Każdy węzeł drzewa reprezentuje fizyczny obiekt urządzenia. W celu dokładnego zapoznania się z zasobami sprzętowymi można skorzystać z opcji *Urządzenia według połączeń* z menu *Widok menedżera urządzeń*.

Biblioteka Setupapi

Biblioteka *Setupapi* zawiera m.in. funkcje (będące odpowiednikami funkcji z modułu *setupapi.h*) związane z instalacją urządzeń. W tabeli 5.4 przedstawiono funkcje pomocne w wyszukaniu konkretnego urządzenia lub klasy urządzeń w systemie i pobierające szczegółowe informacje o interfejsie dostępne z poziomu biblioteki *Setupapi* wraz z ich odpowiednikami z modułu *setupapi.h*.

Tabela 5.4. *Podstawowe funkcje eksportowane z biblioteki Setupapi.dll*

Setupapi.h	Setupapi.dll
`SetupDiGetClassDevs()`	`SetupDiGetClassDevsA()`
`SetupDiEnumDeviceInterfaces()`	`SetupDiEnumDeviceInterfaces()`
`SetupDiDestroyDeviceInfoList()`	`SetupDiDestroyDeviceInfoList()`
`SetupDiGetDeviceInterfaceDetail()`	`SetupDiGetDeviceInterfaceDetailA()`
`SetupDiGetDeviceRegistryProperty()`	`SetupDiGetDeviceRegistryPropertyA()`

Bibliotekę *Setupapi* można łączyć z programem w sposób statyczny, korzystając z modułu *Setupapi.lib* lub dynamicznie wykorzystując *Setupapi.dll*. Należy zwrócić uwagę na fakt, że niektóre (starsze) kompilatory C++ mogą niewłaściwie odwzorowywać identyfikator biblioteki łączonej statycznie w przestrzeni adresowej głównego procesu (programu wykonawczego). Dlatego też dużo bezpieczniejszym sposobem wykorzystania zasobów *Setupapi* jest dynamiczne odwzorowywanie identyfikatora *Setupapi.dll* w przestrzeni adresowej głównego procesu.

Na listingu 5.10 zaprezentowano kod programu posługującego się funkcjami eksportowanymi z biblioteki *Setupapi.dll*. Kod ten jest również ilustracją kolejnego sposobu importowania funkcji z biblioteki dołączanej dynamicznie.

Listing 5.10. *Kod modułu usb_R5_8.cpp*

```
#include <windows>
#pragma option push -a1
#include <setupapi>
#pragma option pop
#include <assert>
#include <iostream>

using namespace std;
```

Kup książkę Poleć książkę

```
void displayError(const char* msg){
  cout << msg << endl;
  system("PAUSE");
  exit(0);
};
//-------------------------------------------------------
template <class T>
inline void releaseMemory(T &x)
{
  assert(x != NULL);
  delete [] x;
  x = NULL;
}
//-------------------------------------------------------
/* A5DCBF10-6530-11D2-901F-00C04FB951ED */
static GUID GUID_DEVINTERFACE_USB_DEVICE =
{0xA5DCBF10, 0x6530, 0x11D2, {0x90, 0x1F, 0x00, 0xC0,
                              0x4F, 0xB9, 0x51, 0xED}};

/* 3ABF6F2D-71C4-462a-8A92-1E6861E6AF27 */
static GUID GUID_DEVINTERFACE_USB_HOST_CONTROLLER =
{0x3abf6f2d, 0x71c4, 0x462a, {0x8a, 0x92, 0x1e, \
                              0x68, 0x61, 0xe6, 0xaf, 0x27}};

/* F18A0E88-C30C-11D0-8815-00A0C906BED8 */
static GUID GUID_DEVINTERFACE_USB_HUB =
{0xf18a0e88, 0xc30c, 0x11d0,{0x88, 0x15, 0x00, \
                             0xa0, 0xc9, 0x06, 0xbe, 0xd8}};
//-------------------------------------------------------
DWORD memberIndex = 0;
DWORD deviceInterfaceDetailDataSize;
DWORD requiredSize;
HMODULE hSetupApi;

HDEVINFO deviceInfoSet;

SP_DEVICE_INTERFACE_DATA deviceInterfaceData;
PSP_DEVICE_INTERFACE_DETAIL_DATA deviceInterfaceDetailData = NULL;

int main(){

    typedef void* (__stdcall *pSetupDiGetClassDevs)
                  (IN LPGUID  ClassGuid,  OPTIONAL
                   IN PCTSTR  Enumerator,  OPTIONAL
                   IN HWND  hwndParent,  OPTIONAL
                   IN DWORD  Flags);

    typedef bool (__stdcall* pSetupDiEnumDeviceInterfaces)
                 (IN HDEVINFO  DeviceInfoSet,
                  IN PSP_DEVINFO_DATA  DeviceInfoData,  OPTIONAL
                  IN LPGUID  InterfaceClassGuid,
                  IN DWORD  MemberIndex,
                  OUT PSP_DEVICE_INTERFACE_DATA  DeviceInterfaceData);

    typedef bool (__stdcall *pSetupDiDestroyDeviceInfoList)(IN void*);

    typedef bool (__stdcall *pSetupDiGetDeviceInterfaceDetail)
                 (IN HDEVINFO  DeviceInfoSet,
```

```
                IN PSP_DEVICE_INTERFACE_DATA  DeviceInterfaceData,
                OUT PSP_DEVICE_INTERFACE_DETAIL_DATA DeviceInterfaceDetailData,
                IN DWORD  DeviceInterfaceDetailDataSize,
                OUT PDWORD  RequiredSize,  OPTIONAL
                OUT PSP_DEVINFO_DATA  DeviceInfoData  OPTIONAL);

 hSetupApi = LoadLibrary("C:\\Windows\\System32\\SETUPAPI.DLL");
 if (!hSetupApi)
   displayError("Błąd dołączenia biblioteki SETUPAPI.DLL.");

 pSetupDiGetClassDevs SetupDiGetClassDevsA = NULL;
 SetupDiGetClassDevsA = (pSetupDiGetClassDevs)GetProcAddress(hSetupApi,
                                              "SetupDiGetClassDevsA");

 pSetupDiEnumDeviceInterfaces SetupDiEnumDeviceInterfaces = NULL;
 SetupDiEnumDeviceInterfaces = (pSetupDiEnumDeviceInterfaces)
            GetProcAddress(hSetupApi, "SetupDiEnumDeviceInterfaces");

 pSetupDiDestroyDeviceInfoList SetupDiDestroyDeviceInfoList = NULL;
 SetupDiDestroyDeviceInfoList = (pSetupDiDestroyDeviceInfoList)
      GetProcAddress(hSetupApi, "SetupDiDestroyDeviceInfoList");

 pSetupDiGetDeviceInterfaceDetail SetupDiGetDeviceInterfaceDetailA = NULL;
 SetupDiGetDeviceInterfaceDetailA = (pSetupDiGetDeviceInterfaceDetail)
      GetProcAddress(hSetupApi, "SetupDiGetDeviceInterfaceDetailA");

 if (!SetupDiGetClassDevsA || !SetupDiEnumDeviceInterfaces ||
     !SetupDiGetDeviceInterfaceDetailA || !SetupDiDestroyDeviceInfoList) {
    FreeLibrary(hSetupApi);
    displayError("Nie znaleziono funkcji eksportowych.");
 }

deviceInfoSet = SetupDiGetClassDevsA(&GUID_DEVINTERFACE_USB_HUB,
                NULL, NULL, DIGCF_PRESENT | DIGCF_INTERFACEDEVICE);
if (deviceInfoSet == INVALID_HANDLE_VALUE)
   displayError("Nie zidentyfikowano podłączonych urządzeń.\n");

deviceInterfaceData.cbSize = sizeof(SP_DEVICE_INTERFACE_DATA);

while(SetupDiEnumDeviceInterfaces(deviceInfoSet, NULL,
                                  &GUID_DEVINTERFACE_USB_HUB,
                                  memberIndex, &deviceInterfaceData)){
    memberIndex++; //inkrementacja numeru interfejsu

    SetupDiGetDeviceInterfaceDetailA(deviceInfoSet, &deviceInterfaceData,
                          NULL, 0, &deviceInterfaceDetailDataSize, NULL);

    deviceInterfaceDetailData = (PSP_DEVICE_INTERFACE_DETAIL_DATA)
                           new DWORD[deviceInterfaceDetailDataSize];

    deviceInterfaceDetailData->cbSize=sizeof(SP_DEVICE_INTERFACE_DETAIL_DATA);

    if (!SetupDiGetDeviceInterfaceDetailA(deviceInfoSet, &deviceInterfaceData,
                   deviceInterfaceDetailData, deviceInterfaceDetailDataSize,
                   &requiredSize, NULL)){
        releaseMemory(deviceInterfaceDetailData);
        SetupDiDestroyDeviceInfoList(deviceInfoSet);
    }
```

```
            cout << deviceInterfaceDetailData->DevicePath << endl;
            releaseMemory(deviceInterfaceDetailData);
        }; //koniec while
        SetupDiDestroyDeviceInfoList(deviceInfoSet);
        cout << endl;
        FreeLibrary(hSetupApi);
        system("PAUSE");
        return 0;
    }
    //-------------------------------------------------------
```

Powiadamianie o dołączaniu i odłączaniu urządzeń

Podłączanie urządzeń LS i FS powoduje zmiany napięcia odpowiednio na liniach D– i D+ (urządzenia HS oraz SS podłącza się tak samo jak FS). Każde urządzenie HS jest na początku traktowane jak urządzenie FS. Dopiero w czasie konfiguracji hub HS sprawdza, czy z tym urządzeniem możliwa jest komunikacja z wysoką szybkością transmisji danych. W podobny sposób jest wykrywane odłączenie urządzenia, z tym że podczas odłączania napięcie na odpowiedniej linii danych maleje do zera, co powoduje zablokowanie portu i sygnalizację zdarzenia w rejestrach statusowych portu i huba.

W trakcie działania programu obsługującego zewnętrzne urządzenie USB istnieje możliwość powiadamiania użytkownika o dołączaniu lub odłączaniu urządzenia. W celu zaprogramowania tego typu funkcjonalności aplikacji w pierwszej kolejności powinniśmy uzyskać interesujący nas identyfikator `GUID` klasy urządzeń. Następnie należy odpowiednio wypełnić (patrz tabela 5.5) pola struktury `DEV_BROADCAST_DEVICEINTERFACE` z modułu *Dbt.h*:

```
typedef struct _DEV_BROADCAST_DEVICEINTERFACE {
  DWORD dbcc_size;
  DWORD dbcc_devicetype;
  DWORD dbcc_reserved;
  GUID dbcc_classguid;
  TCHAR dbcc_name[1];
} DEV_BROADCAST_DEVICEINTERFACE *PDEV_BROADCAST_DEVICEINTERFACE;
```

Tabela 5.5. *Specyfikacja struktury DEV_BROADCAST_DEVICEINTERFACE*

Typ	Element struktury	Znaczenie
`DWORD`	`dbcc_size`	Rozmiar struktury w bajtach plus długość łańcucha znaków wpisanego w polu `dbcc_name` (jeżeli jest używane)
`DWORD`	`dbcc_devicetype`	`DBT_DEVTYP_DEVICEINTERFACE`
`DWORD`	`dbcc_reserved`	Zarezerwowane
`GUID`	`dbcc_classguid`	Identyfikator `GUID` klasy urządzeń
`TCHAR`	`dbcc_name`	Łańcuch znaków (zakończony zerowym ogranicznikiem) określający nazwę urządzenia

Kup książkę Poleć książkę

Kolejnym krokiem jest odpowiednie zarejestrowanie powiadomienia poprzez zadeklarowaną w module *windows.h* funkcję:

```
HDEVNOTIFY WINAPI RegisterDeviceNotification(
  IN  HANDLE hRecipient,
  IN  LPVOID NotificationFilter,
  IN  DWORD Flags
);
```

Parametr `hRecipient` jest identyfikatorem obiektu (np. okna w programie), który otrzyma powiadomienie. Parametr `NotificationFilter` zawiera informacje o urządzeniach, których ma dotyczyć powiadomienie. Znacznik `Flags` przechowuje informacje o odbiorcy powiadomienia. Jeżeli odbiorcą powiadomienia jest okno, znacznik `Flags` przyjmuje wartość `DEVICE_NOTIFY_WINDOW_HANDLE`. W przypadku gdy do `Flags` zostanie wpisana wartość `DEVICE_NOTIFY_ALL_INTERFACE_CLASSES`, powiadomienia będą dotyczyć interfejsów wszystkich klas urządzeń (parametr `dbcc_classguid` jest wówczas ignorowany).

Na listingu 5.11 zaprezentowano kod umożliwiający realizację powiadomień o dołączaniu lub odłączaniu urządzeń klasy HID w trakcie działania programu. Odpowiednie komunikaty są przetwarzane w pętli wywołań funkcji API SDK `GetMessage()`, `TranslateMessage()` oraz `DispatchMessage()` i wyświetlane bezpośrednio na pulpicie (program posługuje się funkcją `WinMain()`). Powiadomienia o zmianie stanu dołączanego lub odłączanego urządzenia USB są generowane w postaci standardowych komunikatów `WM_DEVICECHANGE` w funkcji API SDK:

```
LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg,
                        WPARAM wParam, LPARAM lParam);
```

Wysyła ona komunikat identyfikowany przez `uMsg` do okna identyfikowanego przez parametr `hwnd`.

Listing 5.11. *Kod modułu usb_R5_9.cpp*

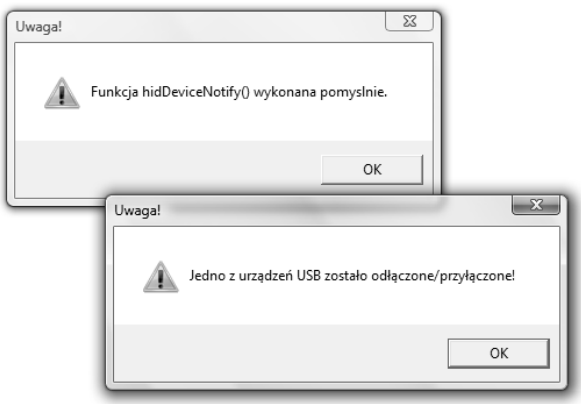
```
#include <objbase.h>
#include <Dbt.h>
#include <windows.h>

static GUID GUID_DEVINTERFACE_USB_DEVICE =
{0xA5DCBF10, 0x6530, 0x11D2, {0x90, 0x1F, 0x00, 0xC0,
                              0x4F, 0xB9, 0x51, 0xED}};

//-------------------------------------------------------
bool hidDeviceNotify(HWND hwnd, GUID GUID_DEVINTERFACE_HID,
                     HDEVNOTIFY *hidDeviceNotify)
{
    DEV_BROADCAST_DEVICEINTERFACE NotificationFilter;
    ZeroMemory(&NotificationFilter, sizeof(NotificationFilter));
    NotificationFilter.dbcc_size = sizeof(DEV_BROADCAST_DEVICEINTERFACE);
    NotificationFilter.dbcc_devicetype = DBT_DEVTYP_DEVICEINTERFACE;
    NotificationFilter.dbcc_classguid = GUID_DEVINTERFACE_USB_DEVICE;
    *hidDeviceNotify = RegisterDeviceNotification(hwnd, &NotificationFilter,
                                          DEVICE_NOTIFY_WINDOW_HANDLE);
    if(!hidDeviceNotify)
```

Kup książkę Poleć książkę

```
    return false;

    return true;
}
//--------------------------------------------------------
LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    switch(uMsg) {
      case WM_DEVICECHANGE:
         MessageBox(NULL, "Jedno z urządzeń USB zostało odłączone/przyłączone!",
                    "Uwaga!", MB_ICONEXCLAMATION | MB_OK);
    break;
      default:
         return DefWindowProc(hwnd, uMsg, wParam, lParam);
    }
    return 0;
}
//--------------------------------------------------------
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE, LPSTR, int)
{
    HWND hWnd;
    WNDCLASSEX wndClassEx;
    HDEVNOTIFY hDeviceNotify;
    MSG Msg;
    wndClassEx.cbSize        = sizeof(WNDCLASSEX);
    wndClassEx.style         = 0;
    wndClassEx.lpfnWndProc   = WindowProc;
    wndClassEx.cbClsExtra    = 0;
    wndClassEx.cbWndExtra    = 0;
    wndClassEx.hInstance     = hInstance;
    wndClassEx.lpszClassName = "DeviceNotifyTest";

    if(!RegisterClassEx(&wndClassEx)){
      MessageBox(NULL, "Błąd rejestracji okna!", "Błąd!",
                 MB_ICONEXCLAMATION | MB_OK);
      return 0;
    }

    hWnd = CreateWindowEx(WS_EX_CLIENTEDGE,"DeviceNotifyTest"," ",WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT, 0, 0, NULL, NULL,
                          hInstance, NULL);
    if(hWnd == NULL){
      MessageBox(NULL, "Błąd utworzenia okna!", "Błąd!",
                 MB_ICONEXCLAMATION | MB_OK);
      return 0;
    }

    if(!hidDeviceNotify(NULL, GUID_DEVINTERFACE_USB_DEVICE, &hDeviceNotify)){
       MessageBox(NULL, "Błąd wykonania funkcji hidDeviceNotify().",
                 "Błąd!", MB_ICONEXCLAMATION | MB_OK);
      return 1;
    }
    MessageBox(NULL, "Funkcja hidDeviceNotify() wykonana pomyślnie.",
               "Uwaga!", MB_ICONEXCLAMATION | MB_OK);

    while(GetMessage(&Msg, NULL, 0, 0) > 0){
      TranslateMessage(&Msg);
```

```
            DispatchMessage(&Msg);
    }
    return Msg.wParam;
}
//--------------------------------------------------------
```

Analiza kodu pozwala zauważyć, że program zostanie uruchomiony w głównym wątku systemu operacyjnego i pozostanie tam *rezydentny* (będzie cyklicznie odbierał komunikaty od sterowników zgodnie z modelem warstwowym systemu USB), dopóki nie zostanie przez użytkownika usunięty z pamięci lub dopóki system operacyjny nie zostanie zrestartowany, tak jak pokazano na rysunku 5.14.



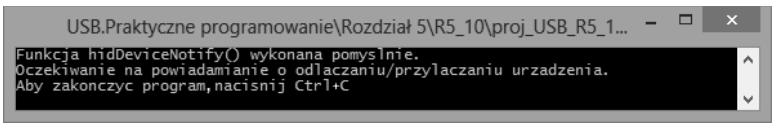
Rysunek 5.14. *Komunikaty o odłączaniu i przyłączaniu urządzeń*

Niedogodności związane z działaniem i obsługą programu opartego na kodzie z listingu 5.10 można rozwiązać w prosty sposób, który polega na zaprogramowaniu interfejsu użytkownika w standardowej funkcji `main()`, tak jak pokazano na listingu 5.12. Na rysunku 5.15 zobrazowano wynik działania tak zmodyfikowanego kodu podczas testów polegających na przyłączaniu i odłączaniu urządzeń USB w trakcie działania programu.

Listing 5.12. *Fragment kodu modułu usb_R5_10.cpp*

```
//--------------------------------------------------------
int main()
{
    const char * const className = "DeviceNotifyTest";
    HDEVNOTIFY hDeviceNotify;
    WNDCLASS wincl = {0};
    wincl.hInstance = GetModuleHandle(0);
    wincl.lpszClassName = className;
    wincl.lpfnWndProc = WindowProc;

    if (!RegisterClass(&wincl)) {
        DWORD error = GetLastError();
        cout << "Błędne wykonanie RegisterClass(), błąd = " << error << endl;
        return 1;
    }

    HWND hwnd = CreateWindowEx(0, className, className,
                               0, 0, 0, 0, 0, 0, 0, 0, 0);
```

```
    if (!hwnd) {
        DWORD error = GetLastError();
        cout << "Błędne wykonanie CreateWindowEx(), błąd = " << error << endl;
        return 1;
    }
    if(!hidDeviceNotify(NULL, GUID_DEVINTERFACE_USB_DEVICE, &hDeviceNotify)){
       cout << "Błąd wykonania funkcji hidDeviceNotify().\n";
      return 1;
    }
    cout << "Funkcja hidDeviceNotify() wykonana pomyślnie.\n";
    cout << "Oczekiwanie na powiadamianie o odłączaniu/"
            "przyłączaniu urządzenia.\n"
            "Aby zakończyć program, naciśnij Ctrl+C\n" << endl;
    for (;;) {
        MSG msg;
        BOOL bRet = GetMessage(&msg, hwnd, 0, 0);
        if ((bRet == 0) || (bRet == -1))
            break;
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return 0;
}
//-------------------------------------------------------------
```



Rysunek 5.15. *Diagnozowanie zdarzeń polegających na przyłączaniu lub odłączaniu urządzeń USB*

Podsumowanie

W niniejszym rozdziale zostały omówione podstawowe zasoby systemowe służące do detekcji oraz identyfikacji interfejsów urządzeń dołączonych do magistrali USB. Zaprezentowano przykłady praktycznego wykorzystania omawianych funkcji i struktur. Konstrukcje przykładowych programów starano się przedstawić w sposób na tyle przejrzysty, by Czytelnik nie miał żadnych problemów z samodzielną ich modyfikacją i dostosowaniem do swoich wymagań sprzętowych i programowych. Starano się również zadbać o czytelność kodu poprzez stosowanie oryginalnego nazewnictwa dla zmiennych i funkcji, które wiernie odzwierciedla ich rolę w programie.

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Skorowidz

Kup książkę Poleć książkę

E

F

Kup książkę Poleć książkę

G

H

I

J

K

L

Ł

M

N

Kup książkę Poleć książkę

O

P

R

Kup książkę Poleć książkę

S

Kup książkę Poleć książkę

Ś

T

U

V

W

Kup książkę Poleć książkę

Z

Kup książkę Poleć książkę

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION

1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

http://program-partnerski.helion.pl

GRUPA WYDAWNICZA
Helion SA

USB

Praktyczne programowanie z Windows API w C++

Mniej więcej pod koniec lat 90. standard USB utrwalił się w świadomości użytkowników komputerów na całym świecie i stał się dla nich jednym z najwygodniejszych narzędzi zapewniających komunikację oraz wymianę danych między urządzeniami. Liczba gniazd USB, do których można podłączyć dosłownie wszystko — mysz, pendrive'a, dysk zewnętrzny czy kartę sieciową — stała się jednym z ważnych kryteriów przy zakupie nowego komputera, a czołowi wytwórcy ani myślą zastępować tego rozwiązania czymkolwiek innym. Jednak USB ma wady. Zalicza się do nich konieczność używania bardziej złożonego sprzętu i oprogramowania w porównaniu ze starszymi protokołami transmisji danych.

Te kłopoty pomoże Ci rozwiązać niniejsza książka, o ile nieobcy Ci jest język C/C++ w zakresie programowania strukturalnego i proceduralnego. Pokaże Ci ona całą architekturę standardu USB oraz implikacje jego stosowania dla różnych urządzeń. Dzięki niej poznasz także podstawy zasad programowania transmisji USB z wykorzystaniem zasobów systemów operacyjnych Windows oraz współistniejących bibliotek programistycznych. W dodatku autor tego wyczerpującego podręcznika nie poprzestaje na suchym wyliczeniu typów danych i funkcji, lecz zamieszcza mnóstwo wskazówek dotyczących konkretnych, działających aplikacji. Jeśli myślisz o programowaniu transmisji danych w USB, nie znajdziesz nic lepszego!

- Standardy bazowe USB 2.0 oraz 3.0
- Informacje o urządzeniach
- Wstęp do transmisji danych
- Urządzenia klasy HID
- Detekcja i identyfikacja różnych klas urządzeń dołączonych do magistrali USB
- Odblokowanie urządzenia do transmisji
- Odczyt i zapis danych
- Biblioteki WinUSB oraz LibUSB
- Podstawy programowania obiektowego transmisji USB
- Wewnętrzne struktury danych
- Podstawy programowania wielowątkowego transmisji USB
- Adaptery USB

Poznaj jeden z najpopularniejszych standardów wszech czasów!

sięgnij po WIĘCEJ
KOD KORZYŚCI

helion.pl
księgarnia internetowa

Nr katalogowy: 11474

Księgarnia internetowa:
http://helion.pl

Zamówienia telefoniczne:
0 801 339900
0 601 339900

Helion

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

Informatyka w najlepszym wydaniu

ISBN 978-83-246-5539-7

Cena: 69,00 zł